版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

仅供非商业用途或交流学习使用

异步图书
www.epubit.com

自制编程语言

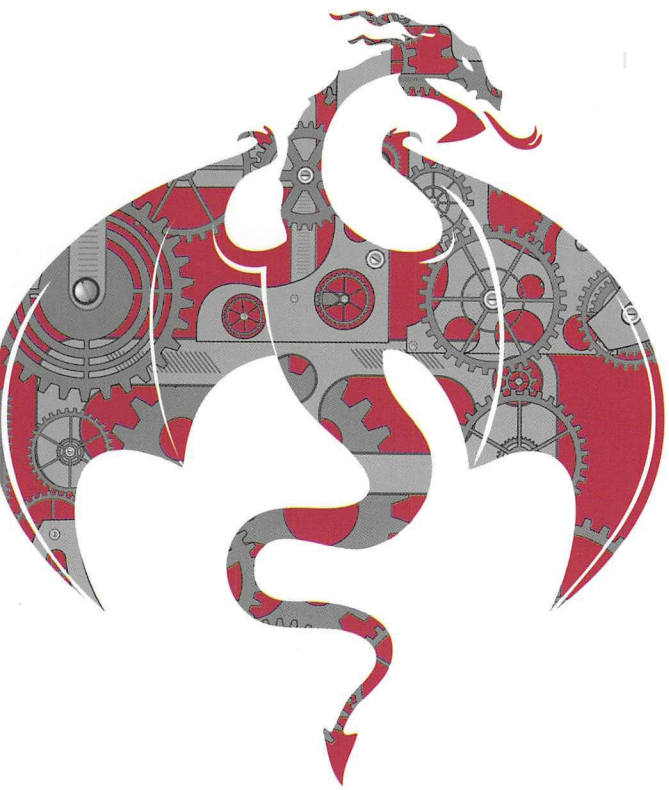
基于C语言

郑钢 著

手把手地教读者从零去实现一门语言，从原理到实践事无巨细

每一步都有实际的代码和详尽的原理说明，读者可以很轻松地掌握各个实现细节

实现脚本语言重要的垃圾回收（GC）、虚拟机运行时和线程等技术都在本书一一呈现



中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

仅供非商业用途或交流学习使用



郑钢：
北京大学硕士，
百度运维研发工程师。
爱父母，
爱老婆，
爱钻研。
微博：郑大刚子，
欢迎互粉。
另著《操作系统真象还原》和《大型网站服务器容量规划》。

欢迎关注
蝙蝠侠学院

仅供非商业用途或交流学习使用

自制编程语言

基于C语言

郑钢 著



人民邮电出版社
北京

仅供非商业用途或交流学习使用

图书在版编目（CIP）数据

自制编程语言 / 郑钢著. -- 北京 : 人民邮电出版社, 2018.9
ISBN 978-7-115-48737-7

Ⅰ. ①自… Ⅱ. ①郑… Ⅲ. ①C语言－程序设计
Ⅳ. ①TP312.8

中国版本图书馆CIP数据核字(2018)第137473号

内 容 提 要

本书是一本专门介绍自制编程语言的图书，书中深入浅出地讲述了如何开发一门编程语言，以及运行这门编程语言的虚拟机。本书主要内容包括：脚本语言的功能、词法分析器、类、对象、原生方法、自上而下算符优先、语法分析、语义分析、虚拟机、内建类、垃圾回收、命令行及调试等技术。

本书适合程序员阅读，也适合对编程语言原理感兴趣的计算机从业人员学习。

◆ 著　　郑　钢
责任编辑　张　涛
责任印制　马振武

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路 11 号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
固安县铭成印刷有限公司印刷

◆ 开本：787×1092　1/16
印张：28
字数：743 千字　　2018 年 9 月第 1 版
印数：1－2 400 册　　2018 年 9 月河北第 1 次印刷

定价：89.00 元

读者服务热线：(010)81055410　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广登字 20170147 号

仅供非商业用途或交流学习使用



推 荐 序

很高兴能成为本书的首批读者，也很高兴能为本书写推荐序。

刚拿到本书手稿时，从书名上我意识到这是对我胃口的书。果然，整书阅读以后，收获颇多。如今程序员的开发成本已经很低了，项目中有各种成熟的框架和库可供选择和使用，但还有人能静下心来研究编译器这么底层的技术，实属难得。本书犹如一把火炬，点燃了技术人内心对开发的热情。

依稀记得 2010 年年初在百度与郑钢初次见面的情景，那时他工作之余的时间基本都用在向各个技术专家请教、讨论各类技术问题上，他是我带过的人中最勤奋的人之一。时光荏苒，一分耕耘一分收获，看到他今天的成长，尤感欣慰。

本书讲述了一门脚本语言（sparrow）的开发过程，这是一本“步步为营”式的书籍，延续了他编写《操作系统真象还原》的风格，手把手地教读者从零实现一门语言，从原理到实践每一步都有实际的代码和详尽的原理说明，通过运行书中各小节中的代码，读者可以很轻松地掌握各个细节，因此本书的学习曲线并不陡峭，甚至很平坦。另外，值得欣喜的是，本书所编写的脚本语言并不是用 Java、C++等入门难度略大的语言实现的，而是用 C 语言，这是我们学习编程的基础语言。也就是说，本书不需要专业的开发经验即可上手学习。另外，在实现过程中并未用到复杂的库函数或系统调用，可以负责地说，本书已经将学习成本降到最低。

C 语言是一种面向过程的语言，如何用一种面向过程的语言去实现一种面向对象的语言很有意思。另外，PHP 和 Perl 语言虽然也实现了类，但它们其实是一种面向过程的语言，并不是纯粹的面向对象语言，而 sparrow 语言是一种纯粹的面向对象语言，它在设计之初就采用对象的方式来处理脚本语言中类的成员和方法，这仿佛让我们看到了面向对象编程语言的基因。众所周知，当今流行的脚本语言应属 Python，Python 也是用 C 语言实现的，也许你很好奇 Python 的内部原理，但是想到它有将近 4 万行的源代码时，也许甚至不想看它的源程序了。那么研读本书中的 sparrow 语言会是一种更好的选择，其源码不足 7100 行，阅读过程轻松愉快，但可以学到 Python 这种语言的实现原理。

对于脚本语言来说，两个重要方面就是垃圾回收和运行环境。垃圾回收就是我们平时所说的 GC（Garbage Collection）。有了 GC，程序员不需要手工释放所分配的对象，可以使精力专注于业务逻辑而不用担心内存泄漏问题。在 sparrow 语言中同样实现了 GC，通过此部分代码你可以看到 GC 的原理，以及哪些对象才能被回收。 运行时环境就是脚本语言中的虚拟机，即 VM（如 Java 语言的 JVM 也是一种 VM）。脚本语言是通过虚拟机才能运行的，如何把编译器生成的操作码转换为实际的代码，这里面的工作对大多数人来说很神秘。相信各位在源码中一探究竟之后会发现：GC 和 VM 这两个神秘的黑盒子不过如此。另外，也许程序员最感兴趣的就是线程，关于线程在用户态下是如何实现的、线程如何实现调度，本书将告诉你答案。总之，但凡涉猎，开卷有益。

每个程序员都有实现属于自己编程语言的梦想，说其是梦想，原因是实现的难度很大.....本书讲的是纯粹的技术“干货”，符合郑钢一贯的写作风格，这是他静心写出来的东西，内容满满，很值得阅读。

于晓声

滴滴系统部技术高级总监



仅供非商业用途或交流学习使用

业界热评

本书详细阐述了设计编程语言所需的基本理论，并且以作者自己开发设计的 sparrow 编程语言为例，引导读者一步一步地实现一门完善的编程语言，本书还讲解了大量的基础类以及垃圾内存回收功能，切实帮助读者从理论过渡到实践，再走向实用。

——肖金亮，阿里蚂蚁金服技术专家

日常有些运维、开发的工作之所以难以开展，很多时候就受限于对底层技术的不了解，本书从独立开发一门编程语言和虚拟机的实践入手，对相关知识进行了阐述，讲解很清楚，实现的技术很值得称赞。

——陈晓聪，百度资深运维工程师

回顾计算机技术发展的这几十年，编程语言层出不穷，语言特性愈发抽象，语言使用则愈发简洁，底层机制隐藏得也越来越深。这种情况下，程序员想深入理解编程语言原理愈发困难，需要花费大量精力去学习语言设计理论并深入阅读语言实现的源码，但往往事倍功半，收效甚微。本书另辟蹊径，带领读者从零开始自己动手实现一个编程语言及其运行环境，循序渐进，在实践过程中透彻理解编程语言的来龙去脉。

——冯顾，360 企业安全集团政企云事业部技术总监

在云与人工智能时代的大背景下，软件从业者都有必要去了解一下虚拟机与一门编程语言。通过对这部分知识的了解能够对操作系统和语言编程有更深刻的理解。市场上介绍编程语言和操作系统入门的书籍不多，而本书是一本较好的读物，很值得读者学习。

——陆景玉，ACFUN 高级运维总监

自制编程语言和虚拟机，这是一个看似很深奥的课题，也涉及当今互联网流行的主题，许多技术人员对其心驰神往，但要领悟其精髓步履维艰。本书循序渐进、由浅到深地讲解了丰富的基础知识，覆盖了常见的编译原理入门知识，更难能可贵的是，作者讲解的知识具有其独特的理解和视角，相信本书能让读者受益匪浅。

——黄梦溪，Mobvista 运维总监

仅供非商业用途或交流学习使用

前　言

很多读者看了我写的《操作系统真象还原》(一本一步步编写操作系统的书)书后，纷纷来信，要求我再写一本自制编程语言的书。这也在情理之中，对于很多计算机从业者来说，操作系统和编译器几乎是两座无法逾越的大山，其难度之大，令很多人员望而生畏。最终，在读者的鼓励下，一冲动就答应了写作本书，其实我很“后悔”做出这样的决定。

为什么后悔呢？因为写书代价很大。

首先，写书相当累，占用很多精力。其次，占用自己学习的时间，在当今个人进步缓慢就算退步的时代，自己没有提升技术会很恐慌。再次，精力全放在写书上会影响家庭、影响工作。最后，还要负责解答许多问题，确实很累。而且，这一次可是在创造编程语言，难度系数太高了，不亚于开发一个操作系统，甚至我父母都劝我：小刚，你都多大了还写书，好好过日子、踏实上班就行了。但是，我最后还是决定写本书。

下面是我跨过重重拦阻创作本书的动机。

1. 有梦想，有远方

既然写书代价那么大，那我为什么还要“明知山有虎，偏向虎山行”呢？因为我就是奔着“老虎”去的，没有老虎的山就没有探险的乐趣。

2. 有难度才有价值

每次遇到一件很难的工作时，我先是“痛苦”，然后随之而来的就是“兴奋”，因为这意味着我要进步。也许读者会说，一定会进步吗？也许 99%会失败。

同样一件事，每个人对它的态度都不同，懦夫看到的是：99%会失败，别干了。勇士看到的是：还有 1%成功的机会，干吧！

只要不放弃（注意，不是坚持），一定会成功，成功只是时间长短的问题。

3. 人生的意义

人生最大的遗憾是“壮志未酬”。如果你是天才，请将自己的才华“挥霍”得一滴不剩，直到触碰到自己智力上的天花板，这样才甘心。如果你是大力士，请努力在奥运赛场上为国争光，直到累得站不起来，这样才甘心。

这正是我写本书的信仰。

学习很累并且无止境，但是多知道一些就会有多一些的欣喜。本着“把自己的知识多掏点给大家”的诚意，本书依然从第 0 章开始，相对《操作系统真象还原》来说，本书的语言不再那么活泼（啰唆）了，毕竟编译器的开发难度略小于开发操作系统，没必要穿插一些“过渡”的话题。本书一步步地实现了一种称为 sparrow 的编程语言，它是用虚拟机运行的，因此最后还要实现一个虚拟机。sparrow 语言是用 C 语言编写的，学习的难度较低，实现的代码不长，希望大家在学习的旅途中愉快。

4. 作者介绍

郑钢，大家都叫我刚子，毕业于北京大学，前百度运维开发工程师，《操作系统真象还原》作

仅供非商业用途或交流学习使用



者。爱父母、爱老婆、爱街舞、爱学习、爱养生。欢迎关注作者的微博“程序员刚子”。

谨以此书感谢我的妻子：有了你对我无微不至的照顾，本书才得以顺利完成。也许以后会有很多人通过本书写出自己的编程语言，成为某某语言之父，这其中也有你的功劳，在这里轻轻地说一声：你辛苦了，我永远爱你！

本书读者 QQ 交流群 148177180，欢迎大家随时沟通。源码下载地址为人民邮电出版社官网。

本书编辑和投稿邮箱为：zhangtao@ptpress.com.cn。

本书适合对编译原理及语言处理器设计有兴趣的读者，以及正在学习相关课程的大中专院校学生。同时，已经学习过相关知识，有一定经验的开发者，也一定能从本书新颖的实现方式中受益良多。

作者



仅供非商业用途或交流学习使用

目　录

第 0 章　一些可能令人迷惑的问题……1

0.0　成功的基石不是坚持，而是“不放弃”……1
0.1　你懂编程语言的“心”吗……2
0.2　编程语言的来历……2
0.3　语言一定要用更底层的语言来编写吗……2
0.4　编译型程序和脚本程序的异同……8
0.5　脚本语言的分类……10
0.6　为什么 CPU 要用数字而不是字符串作为指令……11
0.7　为什么脚本语言比编译型语言慢……11
0.8　既然脚本语言比较慢，为什么大家还要用……12
0.9　什么是中间代码……12
0.10　什么是编译器的前端、后端……13
0.11　词法分析、语法分析、语义分析和生成代码并不是串行执行……13
0.12　什么是符号表……14
0.13　什么是关系中的闭包……14
0.14　什么是程序中的闭包……15
0.15　什么是字母表……16
0.16　什么是语言……17
0.17　正规式就是正则表达式……17
0.18　什么是正规（表达）式和正规集……17
0.19　什么是有穷自动机……18
0.20　有穷自动机与词法分析的关系……19
0.21　词法分析用有穷自动机（有穷状态自动机）的弊端……19
0.22　什么是文法……20
0.23　BNF 和 EBNF，非终结符和终结符，开始符号及产生式……21
0.24　什么是句型、句子、短语……23
0.25　什么是语法分析……24
0.26　语法分析中的推导和归约为什么都要最“左”……25
0.27　什么是语义分析……26
0.28　什么是语法制导……27
0.29　词法分析器吃的是 lex，挤出来的是 token……27
0.30　什么是“遍”……28
0.31　文法为什么可以变换……28
0.32　为什么消除左递归和提取左因子……28
0.33　FIRST 集、FOLLOW 集、LL(1) 文法……29
0.34　最右推导、最左归约、句柄……31
0.35　算符优先分析法……32
0.36　算符优先文法……33
0.37　非终结符中常常定义的因子和项是什么……33
0.38　什么是抽象语法树……33
0.39　编译器如何使用或实现文法中的产生式……34
0.40　程序计数器 pc 与 ip 的区别……35

第 1 章　设计一种面向对象脚本语言……36

1.1　脚本语言的功能……36
1.2　关键字……37
1.3　脚本的执行方式……38
1.4　“纯手工”的开发环境……38
1.5　定义 sparrow 语言的文法……38

第 2 章　实现词法分析器……46

2.1　柔性数组……46
2.2　什么是字节序……47
2.3　一些基础的数据结构（本节源码 stepByStep/c2/a）……48
2.4　定义虚拟机结构（本节源码 stepByStep/c2/b）……56
2.5　实现源码读取（本节源码 stepByStep/c2/c）……57
2.6　unicode 与 UTF-8……59
2.6.1　什么是 unicode……59
2.6.2　什么是 UTF-8……59
2.6.3　UTF-8 编码规则……60

仅供非商业用途或交流学习使用

2.6.4 实现 UTF-8 编码、解码（本节源码 stepByStep/c2/d）……61
2.7 实现词法分析器 parser（本节源码 stepByStep/c2/e）……66
2.7.1 lex 和 token……66
2.7.2 字符串和字符串内嵌表达式……66
2.7.3 单词识别流程……67
2.7.4 定义 token 和 parser……68
2.7.5 解析关键字及获取字符……71
2.7.6 解析标识符和 unicode 码点……73
2.7.7 解析字符串、内嵌表达式、转义字符……75
2.7.8 跳过注释和空行……77
2.7.9 获取 token……79
2.7.10 token 匹配和初始化 parser……84
2.8 构建主程序（本节源码 stepByStep/c2/f）……85
2.9 编译、测试（本节源码 stepByStep/c2/f）……88
2.9.1 一个简单的 makefile……88
2.9.2 测试 paser……92
第 3 章 类与对象……95
3.1 对象在 C 语言中的概貌……95
3.2 实现对象头（本节源码 stepByStep/c3/a）……96
3.3 实现 class 定义（本节源码 stepByStep/c3/a）……99
3.4 实现字符串对象（本节源码 stepByStep/c3/a）……101
3.5 模块对象和实例对象（本节源码 stepByStep/c3/a）……103
3.6 upvalue、openUpvalue 和 closedUpvalue……106
3.7 实现函数对象、闭包对象与调用框架（本节源码 stepByStep/c3/a）……107
3.8 完善词法分析器之数字解析（本节源码 stepByStep/c3/b）……111
3.9 完善词法分析器之字符串解析和获取 token（本节源码 stepByStep/c3/b）……114
3.10 最终版词法分析器的功能验证（本节源码 stepByStep/c3/b）……116
3.11 实现 list 列表对象（本节源码 stepByStep/c3/c）……118
3.12 range 对象（本节源码 stepByStep/c3/c）……121
3.13 迟到的 class.c（本节源码 stepByStep/c3/c）……122
3.14 map 对象（本节源码 stepByStep/c3/c）……124
3.14.1 哈希表……124
3.14.2 map 对象头文件及 entry……125
3.14.3 冲突探测链与伪删除……126
3.14.4 map 对象的实现……128
3.15 线程对象（本节源码 stepByStep/c3/c）……134
3.15.1 线程、协程浅述……134
3.15.2 运行时栈……137
3.15.3 用户线程的实现……138
第 4 章 原生方法及基础实现……142
4.1 解释器流程（本节源码 stepBystep/c4/a）……142
4.2 符号表……144
4.2.1 模块的符号表……144
4.2.2 类方法的符号表……144
4.2.3 模块变量符号表……146
4.2.4 局部变量符号表……147
4.2.5 常量符号表……147
4.3 方法在运行时栈中的参数……147
4.4 定义模块变量（本节源码 stepByStep/c4/b）……148
4.5 原生方法（本节源码 stepByStep/c4/b）……154
4.5.1 定义裸类……154
4.5.2 定义返回值与方法绑定的宏……155
4.5.3 定义原生方法……157
4.5.4 符号表操作……159
4.5.5 定义类、绑定方法、绑定基类……160
4.6 元类及实现（本节源码 stepByStep/c4/b）……161
4.6.1 meta-class 类、class 类、object 类……161
4.6.2 创建元类，绑定类方法……163
4.7 加载模块（本节源码 stepByStep/c4/c）……164
4.8 虚拟机简介……166
4.8.1 虚拟机分类及优缺点……166
4.8.2 为什么要采用虚拟机……168

4.8.3 虚拟机的简单优化 ……170
4.9 字节码 ……171

第 5 章 自上而下算符优先——TDOP ……177

5.1 自上而下算符优先——TDOP ……177
5.2 来自 Douglas Crockford 的教程 ……177
5.3 TDOP 原理 ……194
5.3.1 一些概念 ……194
5.3.2 一个小例子 ……196
5.3.3 expression 的思想 ……197
5.3.4 while（rbp < token.lbp）的意义 ……200
5.3.5 进入 expression 时当前 token 的类别 ……201
5.3.6 TDOP 总结 ……202

第 6 章 实现语法分析与语义分析 ……204

6.1 定义指令（本节源码 stepByStep/c6/a） ……204
6.2 核心脚本（本节源码 stepByStep/c6/a） ……206
6.3 写入指令（本节源码 stepByStep/c6/a） ……212
6.4 编译模块（本节源码 stepByStep/c6/a） ……216
6.5 语义分析的本质 ……218
6.6 注册编译函数（本节源码 stepByStep/c6/b） ……218
6.7 赋值运算的条件 ……221
6.8 实现 expression 及其周边（本节源码 stepByStep/c6/c） ……223
6.9 局部变量作用域管理 ……228
6.10 变量声明、中缀、前缀及混合运算符方法签名（本节源码 stepByStep/c6/d） ……229
6.11 解析标识符（本节源码 stepByStep/c6/e） ……233
6.11.1 处理参数列表及相关 ……233
6.11.2 实现运算符和标识符的签名函数 ……235
6.11.3 upvalue 的查找与添加 ……239
6.11.4 变量的加载与存储 ……242
6.11.5 编译代码块及结束编译单元 ……243
6.11.6 各种方法调用 ……246
6.11.7 标识符的编译 ……249
6.12 编译内嵌表达式（本节源码 stepByStep/c6/f） ……256
6.13 编译 bool 及 null（本节源码 stepByStep/c6/g） ……258
6.14 this、继承、基类（本节源码 stepByStep/c6/h） ……259
6.15 编译小括号、中括号及 list 列表字面量（本节源码 stepByStep/c6/i） ……260
6.16 编译方法调用和 map 字面量（本节源码 stepByStep/c6/j） ……263
6.17 编译数学运算符（本节源码 stepByStep/c6/k） ……266
6.18 编译变量定义（本节源码 stepByStep/c6/l） ……270
6.19 编译语句 ……274
6.19.1 编译 if 语句（本节源码 stepByStep/c6/m） ……274
6.19.2 编译 while 语句（本节源码 stepByStep/c6/n） ……275
6.19.3 编译 return、break 和 continue 语句（本节源码 stepByStep/c6/o） ……280
6.19.4 编译 for 循环语句（本节源码 stepByStep/c6/p） ……284
6.19.5 编译代码块及单一语句（本节源码 stepByStep/c6/q） ……288
6.20 编译类定义（本节源码 stepByStep/c6/r） ……289
6.20.1 方法的声明与定义 ……289
6.20.2 构造函数与创建对象 ……291
6.20.3 编译方法 ……293
6.20.4 编译类定义 ……296
6.21 编译函数定义（本节源码 stepByStep/c6/s） ……298
6.22 编译模块导入（本节源码 stepByStep/c6/t） ……300

第 7 章 虚拟机 ……306

7.1 创建类与堆栈框架（本节源码 stepByStep/c7/a） ……306
7.2 upvalue 的创建与关闭（本节源码 stepByStep/c7/b） ……309
7.3 修正操作数（本节源码 stepByStep/c7/c） ……312

7.4 执行指令（本节源码 stepByStep/c7/d）……314
7.4.1 一些基础工作……314
7.4.2 解码、译码、执行（本节源码 stepByStep/c7/d）……316
7.5 运行虚拟机（本节源码 stepByStep/c7/e）……334
第 8 章 内建类及其方法……337
8.1 Bool 类及其方法（本节源码 stepByStep/c8/a）……337
8.2 线程类及其方法（本节源码 stepByStep/c8/b）……338
8.3 函数类及其方法和函数调用重载（本节源码 stepByStep/c8/c）……345
8.4 Null 类及其方法（本节源码 stepByStep/c8/d）……347
8.5 Num 类及其方法（本节源码 stepByStep/c8/e）……348
8.6 String 类及其方法（本节源码 stepByStep/c8/f）……355
8.7 List 类及其方法（本节源码 stepByStep/c8/g）……369
8.8 Map 类及其方法（本节源码 stepByStep/c8/h）……374
8.9 range 类及其方法（本节源码 stepByStep/c8/i）……380
8.10 System 类及其方法（本节源码 stepByStep/c8/j）……383
8.11 收尾与测试（本节源码 stepByStep/c8/k）……388
第 9 章 垃圾回收……393
9.1 垃圾回收浅述……393
9.2 理论基础……395
9.3 标记—清扫回收算法……396
9.4 一些基础结构（本节源码 stepByStep/c9/a）……397
9.5 实现 GC（本节源码 stepByStep/c9/a）……400
9.6 添加临时根对象与触发 GC……411
第 10 章 命令行及调试……415
10.1 释放虚拟机（本节源码 stepByStep/c10/a）……415
10.2 简单的命令行界面（本节源码 stepByStep/c10/a）……415
10.3 调试（本节源码 stepByStep/c10/b）……417

第0章　一些可能令人迷惑的问题

本章涉及一些编译原理基础，我担心没学过编译原理的读者会觉得吃力，因此顺带介绍了编译原理的基础知识。第1章以后的内容就不需要这些基础了，不会编译原理也无法阻止你成功写出一门脚本语言。因为原理太抽象了，而且为了严谨，理论总是把简单的描述成复杂的。在实践中你会发现，编译器的实现比理解编译器原理容易，你会发现——原来晦涩难懂的概念其实就是这么简单，以至于你是通过实践才懂得了编译原理。毕竟纸上得来终觉浅，绝知此事要躬行。

总之，如果有一些内容不感兴趣或者我解释得不够清楚你也不必着急，这并不影响后面章节的阅读。

0.0 成功的基石不是坚持，而是“不放弃”

人们常说，坚持是成功的“前提”。我说，既然只是前提，这说明坚持也未必会成功。要想成功，人们需要的是成功的“基石”，而不是“前提”，这个基石就是3个字：不放弃。

大部分读者都觉得开发一门编程语言是很难的事，甚至想都不敢想，我担心你也有这个想法，所以特意用这种方式先和你说说心里话：这本书你买都买了，多少发挥点价值才对得起买书的钱，谁的钱也不是白来的。

首先，我并不会为了鼓励大家而大言不惭地说开发语言“其实不难”“很容易”之类的话，相反，这个方向确实很难，而且就应该很难，我想这也正是吸引你的地方，没有难度哪来的价值，“其实不难、很容易”之类的话是对大家上进心的不尊重。

其次，只有在“我也认为很难”的前提下才能保证大部分的朋友能看懂本书。你看，在普通人眼里从A到D，需要有B和C的推理过程，一个步骤都不能少，在天才眼里，A到D是理所应当的事，不需要解释得太清楚，天才认为B和C都是废话，明摆着的事不需要解释。而我不是天才，所以我会把B和C解释清楚。

回到开头的话，为什么说成功的基石不是“坚持”而是“不放弃”呢？这两个词有啥区别？也许有读者说，不放弃就是做着喜欢的事，让自己爱上学习技术。个人觉得这有点不对了，我觉得我更喜欢吃喝玩乐，因为那是生物的本能，选择技术的原因只是我没那么讨厌它，它是我从众多讨厌的事物中选择的最不讨厌的东西。

放弃是为了减少痛苦，坚持是带着痛苦继续前行。“坚持”是个痛苦的词，但凡靠坚持来做的事情必然建立在痛苦之上，而痛苦就会使人产生放弃的念头，这是生物的本能。用“坚持”来“鼓励”自己硬着头皮干，其实已经输了一半，自己认为痛苦的事很难干下去，干不下去的原因是遇到困难时头脑里有“放弃”的念头，如果把这个念头去掉，那么，只要活着，成功无非是时间长短的问题。这个念头其实就是心理预期，“提前”做好心理预期很重要。

总之，不要给自己“可以放弃”的念头，不要让“可以放弃”成为一种选项，把这个选项去掉，那么，只剩下成功。

0.1 你懂编程语言的“心”吗

先来猜猜这是什么？

它是一种人人必不可少，拥有多种颜色、多种外形的物品。

它是一种质地柔软，可使人免受风寒，给予人们温暖的日常物品。

它是一种使人更加美丽，更受年轻女性欢迎的物品。

它是一种用纽扣、拉链或绳带绑定到身体上的物品。

猜到了吗？其实这是对“衣服”的描述。由于我们都知道什么是衣服，因此我们认为以上 4 种描述都是正确的，通过“免受风寒”这 4 个字便有可能想到是衣服。但对于没见过衣服的人，比如刚出生的小孩儿，他肯定还是不懂，甚至不知道什么是纽扣。

什么是编程语言呢？以下摘自百度百科。

（1）“编程语言"（programming language），是用来定义计算机程序的形式语言。它是一种被标准化的交流技巧，用来向计算机发出指令……

（2）编程语言的描述一般可以分为语法及语义。语法是说明编程语言中，哪些符号或文字的组合方式是正确的，语义则是对于编程的解释……

（3）编程语言俗称“计算机语言”，种类非常多，总的来说可以分成机器语言、汇编语言、高级语言三大类。程序是计算机要执行的指令的集合，而程序全部都是用我们所掌握的语言来编写的……

就像刚才我对衣服的描述，以上的 3 个概念，懂的人早已经懂了，不懂的人还是不懂，回答显得很“鸡肋”。因为对于编程语言的理解并不在语言本身，而是在编译器，编译器是编程语言的“心”，而我们很少有人像了解衣服那样了解编译器，因此对于我们大多数人来说只是熟悉了语言的语法，仅仅是“会用”而已。

那什么是编程语言呢？无论我用多少文字都不足以表述精准与全面，因为语言的本质就是编译器，等你了解编译器后，答案自在心中。目前我只能给出同样“鸡肋”的答案——编程语言是编译器用来“将人类思想转换为计算机行为”的语法规则。

0.2 编程语言的来历

世界上本没有编程语言，有的只是编译器。语言本身只是一系列的语法规则，这个规则对应的“行为”才是我们编程的“意图”，因此从“规则”到“行为”解析便是语言的本质，这就是编译器所做的工作。估计大伙儿都知道，如果想输出字符串，在 PHP 语言中可以用语句 echo，在 C 语言中使用 printf 函数，在 C++中使用 cout，这说明不同的规则对应相同的行为，因此语言规则的多样性只是迷惑人的外表，而本质的行为都是一样的，万变不离其宗。并不是“打印”功能就一定得是 print、out 等相关的字眼儿，那是编译器的设计者为了用户使用方便（当然也是为了他自己设计方便）而采用了大伙儿有共识的关键字，避免不必要的混乱。

0.3 语言一定要用更底层的语言来编写吗

有这个疑问并不奇怪，比如：

（1）Python 是用 C 写的，C 较 Python 来说更适合底层执行。

（2）C 代码在编译后会转换为更底层的汇编代码给汇编器，再由汇编器将汇编代码转换为机

器码。

因此给人的感觉是，一种语言必须要用更底层的语言来实现，其实这是个误解。C 只是起初是用汇编语言写的，因为在 C 语言之前只有汇编语言和机器语言。人总是懒惰的，肯定是挑最方便的用，汇编语言好歹是机器语言的符号化，因此相对来说更好用一些，所以只好用汇编来编写 C 语言，等第一版 C 语言诞生后，他们就用 C 语言来写了。什么？用 C 来编写 C？有些读者内心就崩溃了，似乎像是陷入了死循环。其实这根本不是一回事，因为起作用的并不是 C 语言，而是 C 编译器。语言只是规则，编译器产生的行为才是最关键的，编译器就是个程序，C 代码只是它的文本输入。用 C 来编写 C，这就是自举，假如编译器是用别的语言写的，也许你心里就好受一些了。其实只要所使用的语言具有一定的写文件功能就能够写编译器，为什么这么说呢？因为编译器本身是程序，程序本身是由操作系统加载执行的，操作系统识别程序的格式后按照格式读取程序中的段并加载到内存，最后使程序计数器（寄存器 pc 或 ip）跳到程序入口，该程序就执行了。因此用来编写编译器的语言只要具有一定程度的写文件的能力即可，比如至少要具有形同 seek 的文件定位功能，这可用于按照不同格式的协议在不同的偏移处写入数据，因此用 Python 是可以写出 C 编译器的。在这之前我写过《操作系统真象还原》一书，里面的第 0 章第 0.17 小节“先有的语言还是先有的编译器，第 1 个编译器是怎么产生的”，详细地说明 C 编译器是如何自举的，下面我把它贴过来。

首先肯定的是先有的编程语言，哪怕这个语言简单到只有一个符号。先是设计好语言的规则，然后编写能够识别这套规则的编译器，否则若没有语言规则作为指导方向，编译器的编写将无从下笔。第 1 个编译器是怎么产生的，这个问题我并没有求证，不过可以谈下自己的理解，请大伙儿辩证地看。

这个问题属于哲学中鸡生蛋，蛋生鸡的问题，这种思维回旋性质的本源问题经常让人产生迷惑。可是现实生活中这样的例子太多了，具体如下。

（1）英语老师教学生英语，学生成了英语老师后又可以教其他学生英语。

（2）写新的书需要参考其他旧书，新的书将来又会被更新的书参考，就像本书编写过程一样，要参考许多前辈的著作。

（3）用工具可以制造工具，被制造出来的工具将来又可以制造新的工具。

（4）编译器可以编译出新的编译器。

这种自己创造自己的现象，称为自举。

自举？是不是自己把自己举起来？是的，人是不能把自己举起来的，这个词很形象地描述了这类“后果必须有前因”的现象。

以上前 3 个举的都是生活例子，似乎比第 4 个更容易接受。即使这样，对于前 3 个例子大家依然会有疑问：

（1）第一个会英语的人是谁教的？

（2）第一本书是怎样产生的？

（3）第一个工具是如何制造出来的？

其实看到第（2）个例子大家就可能明白了。世界上的第一本书，它的知识来源肯定是人的记忆，通过向个人或群众打听，把大家都认同的知识记录到某个介质上，这样第一本书就出生了。此后再记录新的知识时，由于有了这本书的参考，不需要重新再向众人打听原有知识了，从此以后便形成了书生书的因果循环。

从书的例子可以证明，本源问题中的第一个，都是由其他事物创建出来的，不是自己创造的自己。

就像先有鸡还是先有蛋一样，一定是先有的其他生命体，这个生命体不是今天所说的鸡。伴

随这个生命体漫长的进化中，突然有一天具备了生蛋的能力（也许这个蛋在最初并不能孵化成鸡，这个生命体又经过漫长的进化，最终可以生出能够孵化成鸡的蛋），于是这个蛋可以生出鸡了。过了很久之后，才有的人类。人一开始便接触的是现在的鸡而不知道那个生命体的存在，所以人只知道鸡是由蛋生出来的。

很容易让人混淆的是编译 C 语言时，它先是被编译成汇编代码，再由汇编代码编译为机器码，这样很容易让人误以为一种语言是基于一种更底层的语言的。似乎没有汇编语言，C 语言就没有办法编译一样。拿 gcc 来说，其内部确实要调用汇编器来完成汇编语言到机器码的翻译工作。因为已经有了汇编语言编译器，那何必浪费这个资源不用，自己非要把 C 语言直接翻译成机器码呢，毕竟汇编器已经无比健壮了，将 C 直接变成机器码这个难度比将 C 语言翻译为汇编语言大多了，这属于重新造轮子的行为。

曾经我就这样问过自己，PHP 解释器是用 C 语言写的，C 编译器是用汇编语言写的（这句话不正确），汇编语言是谁写的呢？后来才知道，编译器 gcc 其实是用 C 语言写的。乍一听，什么？用 C 语言写 C 编译器？自己创造自己，就像电影《超验骇客》一样。当时的思维似乎陷入了死循环一样，现在看来这不奇怪。其实编译器用什么语言写是无所谓的，关键是能编译出指令就行了。编译出的可执行文件是要写到磁盘上的，理论上，某个进程，无论其是不是编译器，只要其关于读写文件的功能足够强大，可以往磁盘上写任意内容，都可以生成可执行文件，直接让操作系统加载运行。想象一下，用 Python 写一个脚本，功能是复制一个二进制可执行文件，新复制出来的文件肯定是可以执行的。那 Python 脚本直接输出这样的一个二进制可执行文件，它自然就是可以直接执行的，完全脱离 Python 解释器了。

编译器其实就是语言，因为编译器在设计之初就是先要规划好某种语言，根据这个语言规则来写合适的编译器。所以说，要发明一种语言，关键是得写出与之配套的编译器，这两者是同时出来的。最初的编译器肯定是简单、粗糙的，因为当时的编程语言肯定不完善，顶多是几个符号而已，所以难以称之为语言。只有功能完善且符合规范，有自己一套体系后才能称之为语言。不用说，这个最初的编译器肯定无法编译今天的 C 语言代码。编程语言只是文本，文本只是用来看的，没有执行能力。最初的编译器肯定是用机器码写出来的。这个编译器能识别文本，可以处理一些符号关键字。随着符号越来越多，不断地去改进这个编译器就是了。

以上的符号说的就是编程语言。后来这个编译器支持的关键字越来越多了，也就是这个编译器支持的编程语言越发强大了，可以写出一些复杂的功能的时候，干脆直接用这个语言写个新的编译器，这个新的编译器出生时，还是需要用旧的编译器编译出来的。只要有了新的编译器，之后就可以和旧的编译器说拜拜了。发明新的编译器实际上就是能够处理更多的符号关键字，也就是又有新的开发语言了，这门语言可以是全新的也可以是最初的语言，这取决于编译器的实现。这个过程不断持续，不断进化，逐渐才有了今天的各种语言解释器，这是个迭代的过程。

图 0-1 在网络上非常火，它常常与励志类的文字相关。起初看到这个雕像在雕刻自己时，我着实被感动了，感受到的是一种成长之痛。今天把它贴过来的目的是想告诉大家，起初的编译器也是功能简单，不成规范的，然而经过不断自我“雕刻”，它才有了今天功能的完善。

下面的内容我参考了别人的文章，由于找不到这位大师的署名，只好在此先献上我真挚的敬意，感谢他对求知者的奉献。

要说到 C 编译器的发展，必须要提到这两位大神——C 语言之父 Dennis Ritchie 和 Ken Thompson。Dennis 和 Ken 在编程语言和操作系统的深远贡献让他们获得了计算机科学的最高荣誉，Dennis 和 Ken 于 1983 年赢得了 ACM 图灵奖。

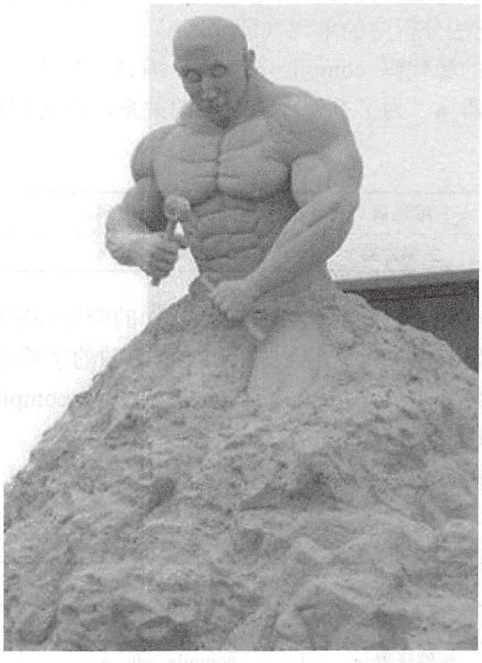


图 0-1

编译器是靠不断学习，不断积累才发展起来的，这是自我学习的过程。下面来看看他们是如何让编译器长大的。

我们都知道转义字符，转义字符是以\开头的多个字符，通常表示某些控制字符，它们通常是不可键入的，也就是这些字符无法在键盘上直接输入，比如\n 表示回车换行，\t 表示 tab。由于以\开头的字符表示转义，因此要想表示\字符本身，就约定用\来转义自己，即\\表示字符\。转义字符虽然表示的是单个字符的意义，在编译器眼里转义字符是多个字符组成的字符串，比如\n 是字符\和 n 组成的字符串。

起初的 C 编译器中并没有处理转义字符，为叙述方便，我们现在称之为旧编译器。如果待编译的代码文件中有字符串\\，这在旧编译器眼里就是\\字符串，并不是转义后的单个字符\。为了表明编译器与作为其输入的代码文件的关系，我们称“作为输入的代码文件”为应用程序文件。尽管被编译的代码文件是实现了一个编译器，而在编译器眼里，它只是一个应用程序级的角色。例如，gcc –c a.c 中，a.c 就是应用程序文件。

现在想在编译器中添加对转义字符的支持，那就需要修改旧编译器的源代码，假设旧编译器的源代码文件名为 compile_old.c。被修改后的编译器代码，已不属于旧编译器的源代码，故我们命名其文件名为 compile_new_a.c，图 0-2 是修改后的内容。

代码 compile_new_a.c

```
1    ...
2    c = next();
3    if(c != '\')
4    return c;
5    c = next();
6    if(c == '\')
7    return '\';
8    ...
```

图 0-2

其中，函数 next()的功能是返回待处理文本（即被编译的源码文件）中的下一字符，强调一下是“单个字符”，并不是记法分析中的单词（即 token）。

用旧编译器将新编译器的源代码 compile_new_a.c 编译，生成可执行文件，该文件就是新的编译器，我们取名为新编译器_a。为了方便理清它们的关系，将它们列入表 0-1 中。

表 0-1

编译器自身源代码	编译器	应用程序源代码	输出文件名
compile_old.c	老编译器	compile_new_a.c	新编译器_a，支持\\

这下编译出来的新编译器_a 可以编译含有转义字符\\的应用程序代码了，也就是说，待编译的文件（也就是应用程序代码）中，应该用\\来表示\。而单独的字符\在新编译器_a 中未做处理而无法通过编译。所以此时新编译器_a 是无法编译自己的源代码 compile_new_a.c 的，因为该源文件中只是单个\字符，新编译器_a 只认得\\。

先更新它们的关系，见表 0-2。

表 0-2

编译器自身源代码	编译器	应用程序源代码	输出文件名
compile_old.c	老编译器	compile_new_a.c	新编译器_a，支持\\
compile_new_a.c	新编译器_a	compile_new_a.c	编译失败

也就是说，现在新编译器_a，无法编译自己的源文件 compile_new_a.c，只有旧编译器才能编译它。再说一下，新编译器_a 无法正确编译自己的源文件 compile_new_a.c 的原因是，compile_new_a.c 中\字符应该用转义字符的方式来引用，即所有用\的地方都应该替换为\\。再回头看一下新编译器_a 的源代码 compile_new_a.c，它只处理了字符串\\，单个\没有对应的处理逻辑。下面修改代码，将新修改后的代码命名为 compile_new_b.c，如图 0-3 所示。

代码 compile_new_b.c

```
1    ...
2    c = next();
3    if(c != '\\')
4    return c;
5    c = next();
6    if(c == '\\')
7    return '\\';
8    ...
```

图 0-3

其实 compile_new_b.c 只是更新了转义字符的语法，这是新编译器_a 所支持的新的语法，下面还是以新编译器_a 来编译新的编译器。

用新编译器_a 编译此文件，将生成新编译器_b，将新的关系录入到表 0-3 中。

表 0-3

编译器自身源代码	编译器	应用程序源代码	输出文件名
compile_old.c	旧编译器	compile_new_a.c	新编译器_a，支持\\
compile_new_a.c	新编译器_a	compile_new_a.c	编译失败
compile_new_a.c	新编译器_a	compile_new_b.c	新编译器_b，支持\\

继续之前再说一下：用编译器去编译另一编译器的源码，也许有的读者觉得很费解，其实你把“被编译的编译器源码”当成普通的应用程序源码就特别容易理解了。上面的编译器代码 compile_new_b.c，其第 3、6、7 行的字符串\\被新编译器_a 处理后，会以单字符\来代替（这是新编译器_a 源码中 return 语句的功能），因此最终处理完成后的代码等同于代码 compile_new_a.c。

现在想加上换行符\n 的支持，如图 0-4 所示。

```
1   if(c == 'n')
2   return '\n';
```

图 0-4

由于现在编译器还不认识\n，故这样做肯定不行，不过可以用其 ASCII 码来代替，将其命名为 compile_new_c.c，如图 0-5 所示。

代码 compile_new_c.c

```
1   ...
2   c = next();
3   if(c != '\\')
4   return c;
5   c = next();
6   if(c == '\\')
7   return '\\';
8   if(c == 'n')
9   return 10;
10  ...
```

图 0-5

用新编译器_a 来编译 compile_new_c.c，将生成新编译器_c，新编译器_c 的代码相当于代码 compile_new_c.c 中所有\\被替换为\后的样子，如表 0-4 所列，暂且称之为代码 compile_new_c1.c，如图 0-6 所示。

代码 compile_new_c1.c

```
1   ...
2   c = next();
3   if(c != '\')
4   return c;
5   c = next();
6   if(c == '\')
7   return '\';
8   if(c == 'n')
9   return 10;
10  ...
```

图 0-6

表 0-4

编译器自身源代码	编译器	应用程序源代码	输出文件名
compile_old.c	旧编译器	compile_new_a.c	新编译器_a，支持\\
compile_new_a.c	新编译器_a	compile_new_a.c	编译失败
compile_new_a.c	新编译器_a	compile_new_b.c	新编译器_b，支持\\
compile_new_a.c	新编译器_a	compile_new_c.c	新编译器_c，间接支持\n

最后再修改 compile_new_c.c 为 compile_new_d.c，将 10 用\n 替代，如图 0-7 所示。

代码 compile_new_d.c

```
...
c = next();
if(c != '\\')
return c;
c = next();
if(c == '\\')
return '\\';
if(c == 'n')
return '\n';
...
```

图 0-7

用新编译器_c 编译 compile_new_d.c，生成新编译器 d，将直接识别\n。同理，新编译器 d 的代码相当于代码 compile_new_d.c 中，所有字符串\\被替换为字符\、字符\n 被替换为数字 10 后的样子，即等同于代码 compile_new_c1.c，如表 0-5 所列。

表 0-5

编译器自身源代码	编译器	应用程序源代码	输出文件名
compile_old.c	老编译器	compile_new_a.c	新编译器_a，支持\\
compile_new_a.c	新编译器_a	compile_new_a.c	编译失败
compile_new_a.c	新编译器_a	compile_new_b.c	新编译器_b，支持\\
compile_new_a.c	新编译器_a	compile_new_c.c	新编译器_c，间接支持\n
compile_new_c.c	新编译器_c	compile_new_d.c	新编译器 d，直接支持\n

编译器经过这样不断地训练，功能越来越强大，不过占用的空间也越来越大了。

0.4 编译型程序和脚本程序的异同

两者最明显的区别就是看它们各是谁的“菜”。两者的共性是最终生成的指令都包含操作码和操作数两部分。

编译型程序所生成的指令是二进制形式的机器码和操作数，即二进制流。同样是数据，和文本文件相比，这里的数据是二进制形式，并不是文本字符串（如 ASCII 码或 unicode 等）形式。如果二进制流按照有无格式来划分，无格式的便是纯粹的二进制流，程序的入口便是文件的开始。另外一种是按照某种协议（即格式）组织的二进制流，比如 Lnux 下 elf 格式的可执行文件。它是硬件 CPU 的直接输入，因此硬件 CPU 是“看得到”编译型程序所对应的指令的，CPU 亲自执行它，即机器码是 CPU 的菜。编译型语言编译出来的程序，运行时本身就是一个进程，它是由操作系统直接调用的，也就是由操作系统加载到内存后，操作系统将 CS:IP 寄存器（IA32 体系架构的 CPU）指向这个程序的入口，使它直接上 CPU 运行，这就是所说的 CPU“看得到”它。总之调度器在就绪队列中能看到此进程。

脚本语言，也称为解释型语言，如 JavaScript、Python、Perl、Php、Shell 脚本等。它们本身是文本文件，是作为某个应用程序的输入，这个应用程序是脚本解释器。由于只是文本，这些脚本中的代码在脚本解释器看来和字符串无异。也就是说，脚本中的代码从来没真正上过 CPU 去执行，CPU 的 CS：IP 寄存器从来没指向过它们，在 CPU 眼里只看得到脚本解释器，而这些脚本中的代码，CPU 从来就不知道有它们的存在，脚本程序却因硬件 CPU 而间接“运行”着。这就像

家长给孩子生活费，孩子用生活费养了只狗狗，家长只关心孩子的成长，从不知道狗狗的存在，但狗狗却间接地成长。这些脚本代码看似在按照开发人员的逻辑在执行，本质上是脚本解释器在时时分析这个脚本，动态根据关键字和语法来做出相应的行为。解释器有两大类，一类是边解释边执行，另一类是分析完整个文件后再执行。如果是第一类，那么脚本中若有语法错误，先前正确的部分也会被正常执行，直到遇到错误才退出；如果是第二类，分析整个文件后才执行的目的是为了创建抽象语法树或者是用与之等价的遍历去生成指令，有了指令之后再运行这些指令以表示程序的执行，这一点和编译型程序是一致的。

脚本程序所生成的指令是文本形式的操作码和操作数，即数据以文本字符串的形式存在。其中的操作码称为 opcode，通常 opcode 是自定义的，所以相应的操作数也要符合 opcode 的规则。为了提高效率，一个 opcode 的功能往往相当于几百上千条机器指令的组合。如果虚拟机不是为了效率，多半是用于跨平台模拟程序运行。这种虚拟机所处理的 opcode 就是另一体系架构的机器码，比如在 x86 上模拟执行 MIPS 上的程序，运行在 x86 上的虚拟机所接收的 opcode 就是 MIPS 的机器码。除跨平台模拟外，通常虚拟机的用途是提高执行效率，因此 opcode 很少按照实际机器码来定义，否则还不如直接生成机器指令交给硬件 CPU 执行更快呢。故此种自定义的指令是虚拟机的输入，即所谓虚拟机的菜。虚拟机分为两大类，一类是模拟 CPU，也就是用软件来模拟硬件 CPU 的行为，这种往往是给语言解释器用的，比如 Python 虚拟机。另一类是要虚拟一套完整的计算机硬件，比如用数组虚拟寄存器，用文件虚拟硬盘等，这种虚拟机往往是用来运行操作系统的，比如 VMware，因为只有操作系统才会操作硬件。

脚本程序是文本字符流（即字符串），其以文本文件的形式存储在磁盘上。具体的文本格式由文本编译器决定，执行时由解释器将其读到内存后，逐行语句地分析并执行。执行过程可能是先生成操作码，然后交给虚拟机逐句执行，此时虚拟机起到的就是 CPU 的作用，操作码便是虚拟机器的输入。当然也可以不通过虚拟机而直接解析，因为解析源码的顺序就是按照程序的逻辑执行的顺序，也就是生成语法树的顺序，因此在解析过程中就可以同时执行了，比如解析到 2+3 时就可以直接输出 5 了。但方便是有限的，实现复杂的功能就不容易了，因为计算过程中需要额外的数据结构，比较对于函数调用来说总该有个运行时栈来存储参数和局部变量以及函数运行过程中对栈的需求开销。因此对于复杂功能，多数情况下还是专门写个虚拟机来完成。

顺便猜想一下解释型语言是如何执行的。我们在执行一个 PHP 脚本时，其实就是启动一个 C 语言编写出来的解释器而已。这个解释器就是一个进程，和一般的进程是没有区别的，只是这个进程的输入则是这个 PHP 脚本。在 PHP 解释器中，这个脚本就是个长一些的字符串，根本不是什么指令代码之类。只是这种解释器了解这种语法，按照语法规则来输出罢了。举个例子，假设下面是文件名为 a.php 的 PHP 代码。

```
<?php                    这是 php 语法中的固定开始标签
   echo "abcd";          输出字符串 abcd
?>                       固定结束标签
```

php 解释器分析文本文件 a.php 时，发现里面的 echo 关键字，将其后面的参数获取后就调用 C 语言中提供的输出函数，比如 printf((echo 的参数))。PHP 解释器对于 PHP 脚本，就相当于浏览器对于 JavaScript 一样。不过这个完全是我猜测的，我不知道 PHP 解释器里面的具体工作，以上只是为了说清楚我的想法，请大家辩证地看。

说到最后，也许你有疑问，如果 CPU 的操作数是字符串的话，那 CPU 就能直接执行脚本语言了，为什么 CPU 不直接支持字符串作为指令呢？后面小节有分享。

0.5 脚本语言的分类

脚本语言大致可分为以下 4 类。

（1）基于命令的语言系统

在这种语言系统中，每一行的代码实际上就是命令和相应的参数，早期的汇编语言就是这种形式。此类语言系统编写的程序就是解决某一问题的一系列步骤，程序的执行过程就是解决问题的过程，就像做菜一样，步骤是提前写好在脑子里（或菜谱中）的。如以下炒菜脚本。

```
WashVegetable  tomato  3   // 洗 3 个西红柿
chopVegetable  tomato  3   // 切 3 个西红柿
heatPot  iron          // 加热铁锅
pourOilIntoPot  50     // 放油 50 克
putVegetable  tomato   // 放入西红柿
stir       //搅拌菜
stir       //搅拌菜
stir       //搅拌菜
……略
```

以上步骤中第 1 列都是命令，后面是命令的参数。其中把菜放进锅后不断地搅拌（示意而已，不用太严谨），由于命令式语言系统中没有循环语句，需要连续填入多个 stir 以实现连续多个相同的操作。会有一个解释器逐行分析此文件，执行相应命令的处理函数。以下是一个解释器示例。

```
#define MAX_CMD_BUF 1024;
char line[MAX_CMD_BUF] = {'\0'};
#define MAX_CMD_ARG 10;
char* fields[1 + MAX_CMD_ARG];  //最多支持 9 个参数的命令
while (getline(line)) {
  //split 是自定义函数，将参数 1 按照参数 2 拆分成多个字段，
  //各字段起始地址放在 fields 数组中，参数 2 被替换为\0
   split(line, '  ',  fields);
   if (memcmp(fields[0], "WashVegetable", strlen(WashVegetable)) == 0) {
       doWashVegetable(fields[1], fields[2]);
   } else if (memcmp(fields[0], "chopVegetable", strlen(chopVegetable)) == 0) {
       doChopVegetable(fields[1], fields[2]);
   } else if (memcmp(fields[0], "heatPot", strlen(heatPot)) == 0) {
       doHeatPot(fields[1]);
   } else if (memcmp(fields[0], "pourOilIntoPot", strlen(pourOilIntoPot)) == 0) {
       doPourOilIntoPot(fields[1]);
   } else if (memcmp(fields[0], "putVegetable", strlen(putVegetable)) == 0) {
       doPutVegetable(fields[1]);
   } else if(memcmp(fields[0], "stir", strlen(stir)) == 0) {
       doStir()
   }
   ……
}
```

（2）基于规则的语言系统

此类语言的执行是基于条件规则，当满足规则时便触发相应的动作。其语言结构是谓词逻

辑→动作，如图 0-8 所示。

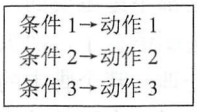


图 0-8

因此此类语言常称为逻辑语言，常用于自然语言处理及人工智能方面，典型的代表有 Prolog。

（3）面向过程的语言系统

面向过程的语言系统我们都比较熟悉，批处理脚本和 shell 脚本，perl、lua 等属于此类，和基于命令的语言系统相比，它可以把一系列命令封装成一个代码块供反复调用。此代码块便是借用了数学中函数的概念，一个 x 对应一个 y，即给一个输入便有一个输出，于是这个代码块便称为函数。

（4）面向对象的语言系统

现代脚本语言基本上都是面向对象，大伙儿用的都挺多的，比如 python。很多读者误以为只要语言中含有关键字 class，那么该语言就是面向对象的语言，这就不严谨了。因为在 perl 语言中也可以通过关键字 class 定义一个类，但其内部实现上并不是完全面向对象，其本质是面向过程的语言。世界上第一款血统纯正的面向对象语言是 smalltalk，它在实现上就是一切皆对象，具有完全面向对象的基因。

0.6 为什么 CPU 要用数字而不是字符串作为指令

在之前小节“编译型程序和脚本程序的异同”的结束处我们讨论过，为什么 CPU 不直接支持字符串作为指令。我估计有的读者会误以为 CPU 将直接执行汇编代码，这是不对的，因为汇编代码是机器码的符号化表示，几乎是与机器码一一对应，但汇编代码绝对不是机器语言。你想，如果汇编代码是机器指令的话，那么 CPU 看到的输入便是字符串，比如以下汇编代码用于计算 1+10-2。

```
mov  a,  1
add  a,  10
dec  a,  2
```

汇编语言其实是汇编器的输入，对于汇编器来说，汇编代码文件也是文本，因此其中 mov 指令也是字符串。如果让 CPU 直接读取汇编文件逐行分析各种字符串以判断指令，这效率必然非常低下。毕竟要比较的字符数太多，比较的次数多了效率当然就低了，因此把指令编号为数字，这样比较数字多省事。而且最主要的是，CPU 更擅长处理数字，它本身的基因就是数字电路，数字计算是建立在数值处理的基础上，这就是本质上二进制数据比文本 ASCII 码更快更紧凑的原因。

0.7 为什么脚本语言比编译型语言慢

而脚本语言的编译有两类，一类是边解释边执行，不产生指令，这个解释过程最占时间的部分就是字符串的比较过程，字符串比较的时间复杂度是 O(n)，也就是在比较 n 次之后解释器才确定了操作码是什么，然后再去获取操作码的操作数，你看能不慢吗？而编译型语言编译后是机器码，是二进制数字，因此可直接上 CPU 运行，而 CPU 擅长处理数字，比较一次数字便可确定操

作码。另一类脚本语言是先编译，再生成操作码，最后交给虚拟机执行，这样多了一个生成操作码的过程，似乎“显得”更慢了。其实这都不是主要的。你看，程序“执行”速度的快慢是比较出来的，编译型语言在执行时已经是二进制语言了，而大多数脚本语言在执行时还是文本，必然要先有个编译过程。这里面全是字符串处理，整个脚本的源码对于编译器来说就是一个长长的字符串，都要完整地进行各种比较，因此多了一个冗长的步骤，必然要慢。有些脚本系统为减少编译的过程，第一次编译后将编译结果缓存为文件，如 Python 会将.py 文件编译后存储为.pyc 文件，下次无须编译直接运行便可。但是，这样无须二次编译的脚本语言就能和编译型程序媲美吗？不见得磁盘 IO 是整个系统最慢的部分，解释器读取缓存文件难道不需要时间吗？等等，有读者说了，编译型的程序被操作系统加载时也要从磁盘上读取啊，这不一样吗？当然不一样，别忘了，脚本程序在执行时先要加载解释器，解释器也是位于硬盘上的文件，只是二进制可执行文件而已，依然需要读取硬盘，然后解释器再去从硬盘上读取脚本语言文件并编译脚本文件。你看，编译型程序在执行时只有 1 个 IO，而脚本程序在执行时有两个，比前者多了 1 个低速的 IO 操作，因此，脚本语言更慢一些是注定的。

0.8 既然脚本语言比较慢，为什么大家还要用

这里的语言是指语言的编译器或解释器，以下简称为语言。

语言慢并不影响整个系统，影响整个系统速度的短板并不是语言本身，目前来说系统的瓶颈普遍是在 IO 部分。语言再慢也比 IO 快一个数量级，并不是语言执行速度快 10 倍后整个系统就快 10 倍，语言慢了，整个系统依然不受影响，这要看瓶颈是哪块儿。这就像动物园运送动物的船超载了，人们不会埋怨某些人太胖了，而是清楚地知道占分量的主要是船上的大象，人的体重和大象根本就不是一个量级。

再说，即使是语言提速后，由于 IO 这块跟不上，依然会被阻塞（由于是脚本语言，这里阻塞的是脚本解释器），而且由于语言太慢而显得阻塞时间更漫长。为什么会阻塞呢？这种阻塞往往是由于程序后续的指令需要从 IO 设备读取到的数据，也就是说程序后面的步骤依赖这些数据，没这些数据程序运行没意义。比如说 Web 服务器先要读取硬盘上的数据然后通过网卡发送给用户，必须获得硬盘数据后，web 服务器进程中那部分操作网卡发送数据的指令才能上 CPU 上执行。由于语言的解释器是由 CPU 处理的，CPU 速率肯定比 IO 设备快太多，因此在等待 IO 设备响应的过程中啥也干不了。操作系统为了让宝贵的 CPU 资源得到最大的利用，肯定会把进程（二进制可执行程序或脚本语言的解释器）加入阻塞队列，让其他可直接运行的、不需要阻塞的进程使用 CPU（阻塞指的是并不会上 CPU 运行，也就是将该进程从操作系统调度器的就绪队列中去掉）。而语言（脚本语言解释器）再慢也比 IO 设备快，因此依然会因为更慢的 IO 而难逃阻塞的命运。也就是说，拖慢整个系统后腿的一定是系统中最慢的部分，而无论脚本语言多慢，IO 设备总是会比语言更慢，因此“影响系统性能”这个黑锅，脚本语言不能背。

另一方面大伙儿喜欢用脚本语言的原因是开发效率高，这也是脚本语言被发明的初衷，很多在 C 中需要多个步骤才能实现的功能在脚本语言中一句话就搞定，当然更受开发人员欢迎了。

0.9 什么是中间代码

很多编译器会将源语言先编译为中间代码，最后再编译为目标代码，但中间语言并不是必需的。中间代码简称 IR（Intermediate Representation），是介于源程序和机器语言之间的语言，有 *N* 元式（如三元式、四元式）、逆波兰、树等形式。目标代码是指运行在目标机器上的代码，与目

标机器的体系架构直接相关，编译器干吗不直接生成目标代码，多这一道程序有什么好处呢？

（1）可以跨平台

由于中间代码并不是目标代码，因此可以作为所有平台的公共语言，从而可通过中间代码实现前后端分离。比如在多平台、多语言的环境下开发可提高开发效率，只要在某一平台上编译出中间代码后，中间代码到目标代码的剩余工作可以由目标平台的编译器继续完成。

（2）便于优化

中间代码更接近于源代码，对于优化来说更直接有效。而且可以在一种平台上优化好中间代码，再发送到其他平台编译为目标机器，提高优化效率。

0.10 什么是编译器的前端、后端

编译器的前后端是由中间代码来划分的，如图 0-9 所示。

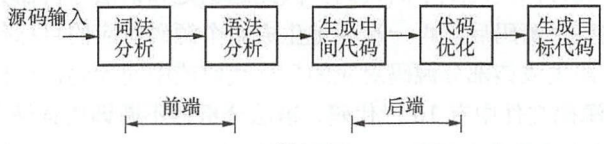


图 0-9

前端主要负责读取源码，对源码进行预处理，通过词法分析把单词变成 Token 流，然后进行语法分析，语义分析，将源码转换为中间代码。

后端负责把中间代码优化后转换为目标代码。

0.11 词法分析、语法分析、语义分析和生成代码并不是串行执行

很多教材上会把编译阶段分为几个独立的部分：

（1）词法分析；

（2）语法分析；

（3）语义分析；

（4）生成中间代码；

（5）优化中间代码；

（6）生成目标代码。

这容易给人造成“这几个步骤是串行执行”的错觉，即“从源码到目标代码必须要顺序地执行这 6 个步骤”，其实不是这样子的，至少一个高效的编译器绝不会这样做。这只是在功能逻辑上的步骤，就拿前 4 步来说，它们是以语法分析为主线，以并行的、穿插的方式在一起执行的，即这 4 个步骤是随语法分析同时开始，同时结束。

每个步骤的功能实现由其实际的模块完成，负责词法分析的模块称为词法分析器，负责生成代码的模块称为代码生成器，负责语法分析的模块称为语法分析器。我们所说的编译器就是由词法分析器、语法分析器和代码生成器组成的（如果有目标代码优化的话还包括优化模块）。编译工作的入口是语法分析，因此编译是以调用语法分析器为开始的，语法分析器会把词法分析器和代码生成器视为两个子例程去调用。换句话说，词法分析器和代码生成器只会被语法分析器调用，如果没有语法分析器，它们就没有“露脸儿”的机会。因此说编译是以语法分析器为主线，由语

仅供非商业用途或交流学习使用

法分析器穿插调用词法分析器和代码生成器并行完成的。

语法分析和语义分析尽管是两个功能，但这其实可以合并为一个。因为在语法分析过后便知道了其语义。这个很好理解，毕竟语法就是语义的规则，规则是由编译器（的设计者）制定的，那么编译器（的设计者）分析了自己设定的规则后当然就明白了语义（不可能不明白自己所制定规则的意义）。比如读英文句子，尤其是复杂的长句，先找到句子谓语动词，以谓语动词为分界线把句子拆分主谓两大部分，在前一部分中找主语，后一部分中找宾语等，在分析完语法后句子的意思就搞清楚了。也就是说，语法分析和语义分析是同时，又是前后脚的事儿，因此合并到一起并不奇怪。你看，语法分析和语义分析确实是并行。

为了语法分析的效率，词法分析器往往是作为一个子例程被语法分析器调用，即每次语法分析器需要一个单词的 token 时就调用词法分析器。你看，语法分析和词法分析确实也是并行。

最后说生成代码。目前生成代码的方式叫语法制导，什么是语法制导呢？就是在分析语法的“同时”生成目标代码或中间代码，实际上就是以语法分析为导向，语法分析器在了解源码语义后立即调用代码生成器生成目标代码或中间代码，因此这也是和语法分析器并行。提醒一下，并不是在语法分析器分析完整个源码后，再一次性地生成整个源码对应的目标代码或中间代码，而是分析一部分源码后就立即生成该部分源码对应的目标代码或中间代码，这样做比较高效且更容易实现。举个例子，比如源码文件中有 10 行代码，语法分析器不断调用词法分析器，每次获得一个单词的 token，把前 3 行源码都读完后确定了源码的语义，立即生成与这 3 行源码同等意义的目标代码或中间代码。然后语法分析器继续调用词法分析器读取第 4 行之后的源码，重复分析语法、生成代码的过程。总之是以语法分析为主线，语法分析把源码按照语法来拆分成多个小部分，每次生成这一小部分的目标代码或中间代码。

总结，为了使编译更加高效，词法分析、语法分析、语义分析和生成代码是以语法分析为中心并行执行的，词法分析和生成代码都是被语法分析器调用的子例程。

0.12 什么是符号表

把符号表列出来是因为这个词听上去“挺唬”人的，由于看不见摸不着，很多初学者都以为它是个非常神秘的东西。其实符号表就是存储符号的表，就是这么简单。你想，源码中的那些符号总该存储在某个地方，这样在引用的时候才能找得到，因此符号表的用途就是记录文件中的符号。符号包括字符串、方法名、变量名、变量值等。符号放在表中的另一个重要原因是便于生成指令，使指令格式统一。编译器会把符号在符号表中的索引作为指令的操作数，如果不用索引的话，指令就会很乱，比如若直接用函数名或字符串作为操作数，指令就冗长了。“表”在计算机中并不专指“表格”，“表”是个笼统的概念，用以表示一切可供增、删、改、查的数据结构，因此符号表可以用任何结构来实现，比如链表、散列表、数组等。

0.13 什么是关系中的闭包

任何事物都会根据其特征来命名，先看看闭包中的“闭”指的是什么。

假如某集合中元素的运算结果还在该集合中，我们称具有这种特征的运算是封闭的，简称闭运算。比如整数集合中的两个整数相加，其结果依然是整数，依然属于整数集合，这就是“闭”的意义。下面先上点甜点。

（1）序偶

序是指顺序，偶是指成对儿。因此序偶是与顺序相关的、一组成对儿出现的两个元素。序偶



仅供非商业用途或交流学习使用

类似于两个元素的集合，但由于与顺序相关，故又不与集合完全相同，比如（小兔，大狗）不等于（大狗，小兔）。

（2）笛卡儿积

有 A 和 B 两个集合，假设某类序偶的第 1 个元素来自集合 A，第 2 个元素来自集合 B，所有这类序偶组成的集合便称集合 A 和集合 B 的笛卡儿积。相当于两个集合中所有元素一一连接，类似数学中多项式相乘，因此笛卡儿积记作 $A\times B$。注意，其中的×是乘号，不是字母 x 的大写形式。

（3）方幂

一个符号串 x 与其自身的 n-1 次连接称此符号串 x 的 n 次方幂，记作 x^n。即 $x^1=x$，$x^2=xx$，$x^3=xxx$……特别地，定义 $x^0=\varepsilon$。

（4）集合的方幂

集合的方幂是集合与自身的连接，如 $A^1=$A，$A^2=AA$，$A^3=AAA$，……，特别地，定义 $A^0=\{\varepsilon\}$。集合的连接运算就是笛卡儿积。若 A 为$\{a,b,c\}$，那么

$A^1=\{a,b,c\}$

$A^2=\{aa,ab,ac,ba,bb,bc,ca,cb,cc\}$

$A^3=\{aaa,aab,aac,aba,abb,abc,aca,acb,acc,...ccc\}$

……

现在可以说闭包了。

闭包：

集合 A 所有方幂的集合称为闭包 A，记作 A^*。

$$A^*=A^0\cup A^1\cup A^2\cup A^3\cup\cdots\cdots$$

闭包 A 去除 A^0 就称为正闭包 A，记作 A^+，也就是说正闭包比闭包少了一个 ε。

0.14 什么是程序中的闭包

程序中的闭包是什么？最直观的解释是闭包是在函数中定义的函数，内部函数引用了外层函数中的局部变量。

程序中的闭包既然也叫“闭”，当然也具备封闭的概念。把什么封闭了呢？封闭的是外部函数作用域中的局部变量，该局部变量称为自由变量。先通过例子建立个感性认识。

```
[work@work tmp]$ cat -n closure.py
     1 def foo ():
     2     vec = [0]
     3     def bar ():
     4         vec[0] += 1
     5         print vec[0]
     6     return bar
     7
     8 foobar = foo()
     9 foobar()
    10 foobar()
    11 foobar()
[work@work tmp]$ python closure.py
1
```

仅供非商业用途或交流学习使用


```
2
3
[work@work tmp]$
```

在 foo 函数内部定义了 bar 函数，bar 函数引用了外部函数 foo 中的 list 元素 vec[0]，对于 bar 函数来说，vec[0]就是自由变量，bar 函数使其自加 1 后打印。但 bar 函数对外并不可见，外部没法直接调用它。为了调用 bar 函数，使 bar 作为 foo 函数的返回值返回给调用者。因此在第 8 行调用 foo 之后返回的是 bar 函数，随后是其输出，符合预期。

有了感性认识后再看下其定义：计算机科学中，闭包（Closure）是引用了自由变量的函数。即使自由变量原来所属的内存空间不存在了，该自由变量也依然与对该函数有效。闭包是函数和其相关的“环境”组成的实体。

我们知道，任何变量都是存在于内存中的，若变量所属的内存被回收了，变量肯定就不存在了，这是常识。可上面定义中说自由变量所属的内存空间不在了（也就是内存空间被回收了）变量也依然有效，看上去这很反常理。因此，要让这一切变得合理的话，只能说明是那个“环境”起了关键作用，并且一定是变量的引用机制对该环境做了很好的屏蔽。用户程序并不知道有那个“环境”的存在，也就是说变量的引用机制会判断：如果自由变量的空间被回收了，就从“环境”中获取变量，否则就去原空间获取。

再说下闭包中的“闭”。闭在此就是封闭，封闭的目的就是为了提供环境。就像院子一样，用院墙封闭起来以构建人的生活环境，在院子里厨房、柴房、厕所都有了，人才能舒适地在里面生存。那闭包为函数提供了什么环境？提供的是自由变量的存储空间。

只有在自由变量原来的空间消失后，编译器才会为该变量提供环境，因此最好先弄明白变量所属的空间为何会消失。这里面涉及局部变量的存储了，我们知道局部变量是存储在栈中的，函数在调用时才会被分配运行时栈，那时才会为函数中的局部变量分配空间。当函数运行结束后，该函数的运行时栈也要被回收，毕竟栈本身占用的空间也在内存中，必须要及时回收以备它用。可是，内部函数引用了外部函数中的局部变量，外部函数调用结束后，外部函数的运行时栈被回收了，位于该运行时栈中的局部变量也就消失了，内部函数引用了那个已经消失的局部变量就会出大问题。闭包的作用就体现在这里，它要为内部函数提供引用该变量的环境，即尽管该变量所在的存储空间消失了，但它额外提供个环境来存储该变量以供内部函数引用，相当于给内部函数“包”了一层环境。该环境是“闭”合的，只供该内部函数使用，因此内部函数便能安心地使用“已经消失”的自由变量。顺便说一句，如何保证自由变量只供内部函数使用？答案是只要保证只有该内部函数能访问到此自由变量就行了。是这样的，代码中定义的函数有很多，为了跟踪各个函数，编译器为每个函数定义了一个结构，这个结构不仅起到标识函数的作用，它也正是我们所说的“环境”，这就是自由变量原空间被回收后的新家。此结构是与函数绑定的，因此保证了只有函数自己独享，即自由变量只供它自己使用。

这下大伙儿也清楚自由变量中“自由”二字的意义了。什么是自由？就是无拘无束不受限制，在此表示变量不受其存储空间的限制，即使它所属的空间被回收了，它也依然还存活。当然，大伙儿都明白这只是“表相”，实际上是编译器给它另外找了个地方安身，在后面实践中你会有更深认识。没学过编译原理的读者最好是按顺序阅读以下内容。

0.15 什么是字母表

字母表是一个有穷符号组成的集合，也就是所能表示的符号的范围，用 Σ 表示。注意，符号并不专指 Φ、λ、Ω 等传统意义的符号，像文字、数字、英文字母等在此都算符号。任何表达式都

仅供非商业用途或交流学习使用

是由符号组成的，符号肯定属于某个集合，表达式也只在该符号集合上有效，字母表就是符号集。比如汉字表达式所属的字母表是中文，英文表达式所属的字母表是英文，汉字不能应用在英文字母表中。如果用程序中的作用域来理解，字母表是表达式的作用域。比如 Σ={x,y,z}，那么在此字母表上的表达式只能用 x、y 和 z 这 3 个字母组成，形式和长度不限，如 xxyyyyz、xyz、yyzx……

0.16 什么是语言

语言是由字母表、文法和语义组成的。

字母表是符号的有穷集合，由字母表中的符号组成单词、句子。文法用来制定单词和句子的种种规则，包括符号在单词中出现顺序的规则、单词在句子中出现顺序的规则等。语义是句子在文法规则下所表达的意图，也就是句子的意义，语义是基于单词和规则，特定的单词在特定的规则下才能表达特定的语义。比如代码 while(exp){循环体}，单词 while 后面必须接单词(，这是由一个字母组成的单词，再后面是循环表达式 exp，这就是“文法”。单词 while 的意思是只要表达式 exp 成立就一直循环执行后面的循环体，这就是“语义”。

后面还会有相关介绍。

0.17 正规式就是正则表达式

网上有人说正规式和正则表达式不是一回事儿，我觉得这么说有些欠妥。

正规式就是正则表达式，两者都是用于模式匹配，很多编译原理教材都有说明，如清华大学出版社出版的《编译原理（第 2 版）》（张素琴、吕映芝、蒋维杜、戴桂兰编著）第 52 页 4.2.2 节倒数第 6 行：正规式也称正则表达式。因此两者是一回事儿。

正规式是个规范、协议，而我们平时所用的正则表达式是这种规范协议的具体实现（这就像 C 语言有很多种编译器一样，C 语法就是规范，每一种编译器就是一种实现），只不过实现的功能更丰富，以至于无法和编译原理教材上所说的正规式对上号了，可以理解为：正规式是最原始的正则表达式。

0.18 什么是正规（表达）式和正规集

正规式也称正规表达式，就是表达“字符排列顺序”的公式。既然称为公式，那么正规式中肯定有运算符，没错，它正是用 3 个运算符来表达字符的排列顺序，如表 0-6 所列。

表 0-6

运算符	文字描述	读作	操作	优先级	示例
|	竖线	或	选择	低	a=x|y 表示 a=x 或 a=y
.	小圆点，可省略	连接	连接	中	x.y 表示 xy
*	星号	闭包	0 次或多次的自我连接	高	a=x*表示 a=ε， 或 a=x，或 a=xx， 或 a=xxx，或 a=xxxx…

以上的 ε 表示空，在此表示 0 次连接。多次的自我连接也称为方幂，就像数学中的幂一样，后面有介绍。

反过来说，由有限次使用这 3 种运算符得到的表达式就是正规式，假设正规式中的符号都属

仅供非商业用途或交流学习使用

于字母表 Σ，那么能用此正规式匹配的所有表达式组成的集合称为字母表 Σ 上的正规集。

概念为了严谨就难免会抽象，简单来说，正规集是正规式所能匹配的一切表达式的集合。举个例子，假设字母表 Σ={a,b}，对于正规式 ba*来说，其正规集是以 1 个 b 开头，后接 0 个或多个 a 的字符串。解释：*表示 0 次以上（包括 0 次）的重复连续出现（即连接），能被 ba*匹配到的表达式有 bε、ba、baa、baaa……等，这些就是 a*的正规集。那么对于正规式 aaab*来说，其正规集是以 3 个 a 开头，后接 0 个或多个 b 的字符串，同理不解释。

0.19 什么是有穷自动机

有穷自动机（Finite Automata）也称有穷状态自动机，是一种数学模型，也称有限状态自动机，这种模型对应的系统具有有限个数的内部状态，系统只需要根据当前所处的状态和面临的输入就能够决定系统的后继行为。当系统处理了输入后，状态也会发生改变。

有穷自动机是由两位神经物理学家 MeCuloch 和 Pitts 提出的概念模型，他们认为人脑也是个有穷状态的系统，尽管状态数目很大，但依然是有穷的。他们所提出的有穷自动机的模型是由 3 部分组成：一条有穷长度的输入带、一个读头和一个控制器，如图 0-10 所示。

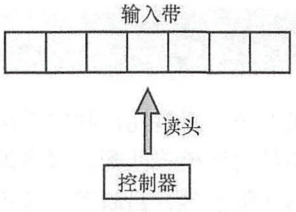


图 0-10

在这个模型中，单个输入称为输入符号，输入带用来存放输入串，每个输入符号占据一个方格，输入带的长度和输入串长度相同。控制器的状态数是有限的，读头只能只读输入带，控制器控制读头从左向右逐个读入每个符号，控制器根据当前状态和输入符号控制转移到的下个状态。

一个有穷自动机包括下列内容。

（1）一个有限的状态集合。

（2）一个有限的字母表。

（3）一个“状态+输入字符”的转移集合，以字母表中的符号作为输入字符。

（4）一个开始状态，也称初态。

（5）一个被称为“可接受状态的状态子集”，可接受状态也称为终态。

如图 0-11 所示。

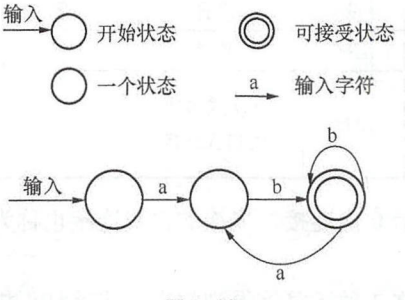


图 0-11

仅供非商业用途或交流学习使用

有穷自动机分为两大类：确定的有穷自动机（Deterministic Finite Automata）和不确定的有穷自动机（Nondeterministic Finite Automata），它们分别是常说的 DFA 和 NFA。

不再介绍，已经够用了。

0.20 有穷自动机与词法分析的关系

先回答，这两者间没直接关系。

编译原理教材中经常把词法分析器和有穷自动机一起介绍，以至于很多兄弟认为词法分析器就是有穷自动机，这样理解欠妥。

有穷自动机也称有限状态自动机，是一种识别装置，“可以”用来识别正规式，也就是说可以识别正规集。注意啦，说的是“可以”用来识别正规式，并不是只用来识别正规式，“可以”具有选项、可选的意思，这说明，词法分析器是把有穷自动机当成识别单词的工具，也就是说，有穷自动机只是词法分析器的一种实现。

计算机科学中可以找到很多有穷状态系统的例子，比如 TCP/IP 网络连接的状态：syn_sent、time_wait、close_wait 等，还有本文中的词法分析器，总之，有穷自动机是设计这些系统的理论工具，也可以说是一种算法。

词分析器为什么要采用有穷自动机来实现单词识别呢？

词法分析器是按照文法的规则来识别单词，文法是人设计的，同样，人在设计词法分析器时也要按照人自己制定的文法规则来识别单词，这是人与自己协调好的。比如文法规定整数是数字字符的正闭包，也就是由一个以上的数字字符连接而成的，浮点数是由整数内部加个小数点表示的，即“整数.整数”为浮点数。人也按照这个“思路”去实现识别，步骤如下。

（1）词法分析器启动后，若输入字符是数字，词法分析器就进入“处理整数的阶段”。

（2）当词法分析器准备处理整数时，如果下一个输入字符还是数字，继续处于“处理整数的阶段”。

（3）当词法分析器处理整数时，如果输入字符是小数点，词法分析器进入“处理浮点数的阶段”。

（4）当词法分析器处理整数时，如果输入字符不是数字或小数点，词法分析器处理整数完成。

（5）当词法分析器处理浮点数时，如果输入字符不是数字，词法分析器处理浮点数完成。

你看，我这里强调了“思路”，即表示先有的“想法”，然后再涉及如何“实现”。词法分析设计者在脑子中构思这种思路时，觉得“处理 xxx 的阶段”很符合有穷自动机的“状态”，输入字符的意义也和有穷自动机一致，“处理整数的阶段”进入“处理浮点数的阶段”完全符合有穷自动机的状态转移，很自然地，就用有穷自动机这个现成的理论工具来帮助实现词法分析器。有穷自动机是一种识别装置，把它应用在词法分析中，可以用来识别正规式所匹配的所有表达式，也就是说可以识别正规集。开始时自动机所处的状态是初态，当自动机从初态进入终态时，输入带上读过的字符串即是自动机所匹配的字符串，正好完成了词法分析的需要。因此，有穷自动机是被用来实现词法分析的算法，属于工具，无论是工具还是算法也都只是一种选择，方法并不是唯一的，因此也可以不采用它。你会看到在第 2 章实现的词法分析器，我们并没有使用有穷自动机。

0.21 词法分析用有穷自动机（有穷状态自动机）的弊端

有穷自动机也称有穷状态自动机，它是编译原理教材上实现词法分析器的经典方案，以下简



仅供非商业用途或交流学习使用

称状态机。

状态机的特点是，从当前状态接收一个字符后马上转移到下一个状态，即使下一个状态就是当前状态，转移的过程也要发生。比如图 0-12 用来识别数字的状态机。

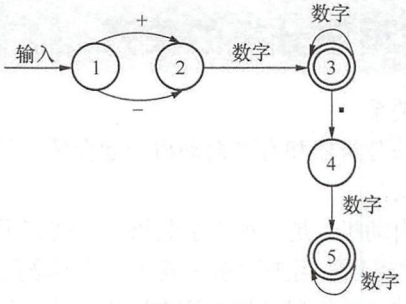


图 0-12

从输入开始，状态机进入状态 1，在读入正负号后状态机转移到状态 2，这是有符号数的开始。下一个接收的是数字，状态机就从状态 2 转移到了状态 3，状态 3 是双圆圈表示的终态，终态即终止态，也称可接受状态，表示状态机接受了所识别到的字符串。在状态 3 时接受任何一个数字都会使状态机重新转移到状态 3，即原地转移到当前状态，这是一个循环过程，故可接受无限个数字。根据此状态转移图，可接受的字符串形式为“+多个数字字符”或“−多个数字字符”，比如+1234 或−123456 等等。到此识别的是整数。如果下一个字符是小数点，那么状态机就从状态 3 进入到浮点数状态 4，继续读入数字，进入状态 5，状态 5 同样是终态，在此状态下可以接受无限个数字。

但实际上，现代词法分析器很少完全用状态机来识别单词，原因如下。

（1）效率低，需要重复进入状态分支。

状态转移的执行过程有冗余，因此效率低下。标准的状态机是每次读入 1 个字符后马上转移到某个状态，即使是状态原地不动也需要走一次这个转移状态的流程，多执行了一些不必要的操作。就拿上图右上角的状态 3 来说，如果严格地按照状态机的思路来编写程序，在读入 1 个数字后，下一步工作不是继续读入数字，而是把状态重复设置为状态 3（实现上也可不必，可理论上是这样的），然后进入状态的流程分支，再读入下一个字符，再继续判断接受此输入后的转移状态，尽管状态没变（还是状态 3），依然再重复上述过程。

（2）需要设置的状态太多，编码太累。

状态是由文法来决定的，文法复杂状态就会复杂，状态太多时编码会烦死的，就连单字符的单词（单词泛指文法中所能推导出的一切终结符，比如+也会被词法分析器视为单词）也要为其专门设置个状态，还不够麻烦的呢，而且这种单字符单词还特别多，比如（、）、{、} 和大部分单字符运算符等。也许仅凭我一面之词不足以让你信服，我当初也不相信，后来自己弄了一个有穷状态自动机版本的词法分析器，弄到一半我就信了。

因此我们的词法分析器也没用状态机，后面会讲实现部分。

0.22 什么是文法

如果一首曲子你不能哼唱给别人听，你该如何向别人描述这首曲子？答案就是用乐谱。同样，如果你设计了一门语言，如何向别人描述其用法？答案就是用文法。文法就是描述语言语法规则的方法。

仅供非商业用途或交流学习使用

文法即语言的语法，是由其产生式来描述的，产生式是语言的规则，因此在语法分析时，就是按照文法产生式去匹配所输入的代码。

“上下文”是指所处的环境。一个上下文无关文法是指文法所定义的语法范畴与上下文无关，完全独立于语法范畴所处的环境，简单来说就是“就事论事”，不必考虑人情世故。比如数学运算就是上下文无关。上下文有关就是指语义要取决于语（环）境。比如老妈说了句“几点了”，这句话在早上说的意思有可能是“还不起床，你想睡到什么时候”；而在晚上说的意思可能是“这么晚了你老爸怎么还不回来”。除非程序员要求代码的意义必须要结合上下文环境才能确定，否则编译器采用上下文无关文法就足以应付目前的编程需求了。上下文无关文法所对应的语言称为上下文无关语言，无特别指出的情况下，本书中所涉及的文法均指这种上下文无关文法。

文法是由一个四元组表示的，元组就是用小括号括起的元素对儿，四元组就是 4 个用括号括起的元素。文法本身是用字母 G 表示，文法内容用一个四元组表示，包括非终结符 Vn，终结符 Vt，产生式 P，开始符号 S，即 G=(Vn, Vt, P, S)。文法是一套有限的规则，但是它可以创建无限符合此规则的句子，这就是产生式的作用，有关产生式请看下回分解。

0.23 BNF 和 EBNF，非终结符和终结符，开始符号及产生式

这都是辅助解释文法的概念。

BNF（Backus-Naur Form），即巴科斯范式，是一种程序设计语言描述工具，是由 John Backus 和 Peter Naur 引入的一种形式化符号，用来描述给定语言的语法，即描述语言的语言，故可以称之为元语言。它的内容如表 0-7 所列。

表 0-7

元符号	含义	备注
->	定义为	也可用:=和::=表示
<>	用<>括起表示非终结符	通常用大写字符代替
“”	双引号中的内容表示终结符	通常用小写字符代替
|	或	用于候选式

由于它过于简单，所以在实际应用中暴露了不足。

（1）循环和可选项不能直接表示。

比如若想表示多个'a'循环，就要写成这样：P->aP。

若想表示可选项（即可以有也可以没有的项），就要写成 P->a|ε。其中 ε 表示空，即没有。

（2）不利于编写语法分析器。

下面为解释缘由会提及一些未介绍的概念，后面有详细介绍，不懂的读者不要急。

由于语法分析器的实现是要根据产生式来编写，所以用 BNF 写出的产生式也不利于程序实现，比如下面描述 if-else 语句的产生式：

IF_STAT -> if (EXP) {STAT} | if (EXP) {STAT} else {STAT}

语法分析器在执行非终结符 IF_STAT 对应的函数时，发现这两个候选式前面的“if (EXP) {STAT}”部分都一样，必需二选一，因此如果“死板地”按照此产生式来分析语法，必然要冒险读入大量的字符串，如果候选项判断错了，就要回溯，回溯就是回退到之前的分叉口进行重新选择，这代价太大了，因此根据这样的产生式来编写语法分析器很不容易。因此有了 EBNF，即扩展 BNF，内容如表 0-8 所列。



仅供非商业用途或交流学习使用

表 0-8

元符号	含义	备注
->	定义为	也可用:=和::=表示
<>	用<>括起表示非终结符	通常用大写字符代替
“”	双引号中的内容表示终结符	通常用小写字符代替
|	或	用于候选式
{}	0 次或多次重复出现	
[]	0 次或 1 次重复出现，即有或没有	
()	括号内的看成一项	

有了 EBNF，之前的循环、可选项及语法分析器的编写就容易了。

循环可写为：

P->a{a}

可选项可写为：

P->[a]

语法分析器所依赖的产生式可写为：

IF_STAT -> if (EXP) {STAT} [else {STAT}]

这样分析器在处理 if (EXP) {STAT}后，只要再从用户代码中判断是否有关键字 else 就可以了，只需要读取 1 个记法单元就可以判断了。

也许有读者会说：“对于以上的 IF_STAT 来说，无论是用 BNF 还是 EBNF 描述，其所描述的语法结构都是一样的，只不过看上去 EBNF 更方便，而 BNF 笨拙一点。我按照 if-else 的语法结构来识别也是一样，因此用 BNF 文法也行，只要我不死板地按照它的描述去编写语法分析器就行了”。道理是这样的，但你想想看，文法的目的是什么？就是用来向人们描述语法规则的，如果不看这个 IF_STAT 的文法，你是怎么知道关键字 if 后面要接左小括号“(”，甚至还可以接一个 else？还不是看了产生式 IF_STAT 后才知道的吗。而且，文法就是编写语法分析器的导航，按照导航去实现语法分析器肯定是最省事儿的。这么说吧，尽管演奏家已经把曲子背下来了，但有个乐谱摆在那，是不是演奏起来更得心应手。所以，文法是为了语法分析器服务的，让语法分析器的设计者更舒服，那才是最好的，否则非要自作聪明跨过文法，按照自己的理解去实现语法分析器，那还要文法干吗呢。

文法是利用“有限”的规则，通过“推导”的方式产生“无限”多的句子来描述某种语言所有合法的使用情况，这里的推导加了引号，非终结符和终结符便是针对这个“推导”来定义的。推导是推出、导出的意思，即用另一种与非终结符等价的表达式替换非终结符，将此非终结符“展开”。终结符是指“推导过程”终止、终结后的产物，也就是推导到头儿了，符号没法再展开了。非终结符是指推导过程还要继续，待推导的符号还没展开彻底，下面再细说说。

终结符通常用 Vt 表示，即 Vocabulary terminated，也就是推导“终结”的意思，这也正是终结符的意义，因此终结符相当于“常量”，通常为字面量，是某个具体、固定的值，比如“(”，终结符用来作为非终结符的值。

非终结符通常用 Vn 表示，即 Vocabulary not terminated。顾名思义，不终结的符号，是指推导可继续进行、还没有推导彻底，它就像变量，变量的值还可以用变量（其他变量或者自身）来表示，那么非终结符也可以用非终结符（其他非终结符或者自身）来表示，还可以用终结符来表示。

无特别指出，本书中终结符用小写字母表示，非终结符用大写字母表示。

语法分析时总该有个语法规则的起始入口，这个入口就是开始符号。开始符号通常用 S 表示，即 Start。它是文法中最大的非终结符，它可以推导出任何其他非终结符及终结符，因此是文法开始推导的入口，就像 DNS 的根域，从最上层开始总能到达任意下层。

文法中的规则称为产生式，即用来“产生”句子的表达式。通常用 P 表示，即 Production，比如规则 E->E+T 就是产生式，->表示定义为。->左边的部分称为产生式的左部，->右边的部分称为产生式的右部。非终结符就是用来“推导”的，就是用其右部等价的表达式替换，将非终结符“展开”为右部。推导是用=>表示，即可推出、可导出的意思，1 步推导也称为直接推导，用符号=>表示，1 步以上推导用 $\overset{+}{\Rightarrow}$ 表示，0 步以上推导用 $\overset{*}{\Rightarrow}$ 表示。如果一个非终结符可直接同时推导出多个表达式，这几个表达式之间属于并列的关系，这几个并列的表达式便称为该终结符的候选式，只有“并列”才能“同一级别”嘛，这样才有资格成为“候选”。也就是说，在产生式右部，若有多个用|分隔的表示“或”的并列的表达式，这几个表达式称为左部的候选式。候选嘛，随时待命等候被选择，如同候选人一样，若是只有 1 个肯定就没得选了，因此多于 1 个可供推导的表达式才称得上“候选”。比如 E->E+T|F。E+T 和 F 都是 E 的候选式。

为什么称为“产生式”呢？这是因为它是“产生”语法的式子。根据某一语法规则（即文法）我们可以写出无限多的代码，比如以下文法 1，如图 0-13 所示。

```
句子  ->  主语 谓语
主语  ->  名词 | 代词
谓语  ->  动词 [宾语]
宾语  ->  名词 | 代词
```

文法 1

图 0-13

|表示或的意思，表示选择其一，|两边的部分称为候选式。[]表示中括号内的部分可出现 0 次或 1 次，即有或没有。按照此方法我们可以写出很多的句子，比如“我走了”“我吃早餐”“我笑了”“我去游泳”，总之可以“产生”无限多的句子，这就是称为“产生”式的原因。

这里补充一下，在实际生产环境中为方便编码会采用“正规文法”来定义语法，正规文法简单来说就是利用正则表达式定义的文法，产生式全是用正则表达式来描述的，咱们在后面也会采用正规文法，因此大伙儿最好有正则表达式的基础。

0.24 什么是句型、句子、短语

从一个文法的开始符号 S 可推导出的任意符号串 a 都称为句型，即 S $\overset{*}{\Rightarrow}$ a，当 a 中全是终结符时，a 就称为句子。

既然称之为句型，就应该按照句型的意义去理解。在英文里，句型是个句子的模板，是可以套用的句式，所以在编译原理中的句型也应具有模板的意义，这里的模板变量就是指非终结符。当句型中没有可替换的部分时，即非终结符全部展开为终结符了，也就成了最终的句子。句子是特殊的句型，相当于句型的实例化。

比如在文法 1 中，“句子 -> 主语 谓语”，其中的“句子”便是开始符号，非终结符“主语”的产生式是“主语 -> 名词 | 代词”，主语有两个候选式，这里选择用非终结符“名词”表示主语，故“句子 -> 主语 谓语”则变为“句子 -> 名词 谓语”，这就是句型。而“我去游泳”便是句子，因为这是把所有非终结符都展开后的结果，里面全是终结符了。

什么是短语呢？先对比一下，句子和句型都是“整”句，这里强调的是完整一句，那么自然

而然，短语就是指句子或句型的一部分，如同英语中的短语是一样的概念，都是句子的组成部分且不能单独存在。因此短语是这样定义的：在文法 G 中，S 是开始符号，假设 $S \overset{*}{\Rightarrow} \alpha A \delta$ 且 $A \overset{+}{\Rightarrow} \beta$，则称 β 是句型 αβδ 相对于非终结符 A 的短语。如果 A=>β，即 A 直接推导出 β，则称 β 是句型 αβδ 相对于产生式 A->β 的直接短语。举个例子，图 0-14 文法中的 3 个产生式。

```
“句子  ->  主语 谓语”
“主语  ->  名词 | 代词”
“名词  ->  小明”
```

图 0-14

主语可直接推导出“名词”，那么“名词”就是句型“句子 -> 名词 谓语”相对于非终结符“主语”（即产生式“主语 -> 名词 | 代词”）的直接短语。强调下，短语是句型或句子的一部分，并不是完整的句子或句型，比如这里的短语是“名词”，要不干吗称为“短”语呢。“名词”其实也是非终结符，可以指代人名、物名等等，这里仅用人名“小明”作为产生式“名词”的右部，而“小明”也就是终结符了，没法再推导了。短语也可以是终结符，比如终结符“小明”是句型“句子 -> 小明 谓语”相对于非终结符“名词”的短语。

注意一，直接短语一定是非终结符的完整右部，如果右部中有多个候选式时就是其中的一个完整的候选式，这一点很重要，尤其是在句柄中，短语就是用来引出并解释句柄的概念，后面会介绍句柄。

注意二，文法就是用来创建句子的，对于语言（包括人类语言和计算机语言）来说句子才是唯一有用的部分，因为文法中的产生式和句型只是构建句子的规则、规律，这样就可以用有限的表达式来构造出无限的句子。总之方法并不是为了创建句型，句型是无用的，句型只是句子的模板，模板的实例化才有用，就像类和对象的关系，类是对象的模板，我们定义了类，目的是为了生成实例对象。人类语言也是一样，人们口中说出来的都是句子，不是句型，句子才能交流，句子才是最具体实在有用的，句型只是个句子“框架”。我们写的源码全是句子，语法分析也都是在分析句子，而句型只是语法分析的中间态，句型是向人们描述：符号应该怎样排列才是正确的句子。

0.25 什么是语法分析

语法分析就是按文法的产生式去识别输入串是否为文法的一个句子，这个输入串就是用户的源码，即语法正确的用户源码就是该文法的句子。

开始符号是文法中最大的非终结符，是文法定义的入口，因此是文法的最“上”层。文法是利用有限的语法规则来创建无限多的句子（源码串），句子是全部由终结符组成的字符串，是由上层的非终结符推导出来的、符合语法的最具体的部分，也就是没法再推导了，是最“下”层，我们平时所写的代码就是最下层的句子。你体会到了吗，语言的语法就那么几条，但我们可以写出千变万化无限多的代码，这就是文法规则的“以有限为无限”。故，语法分析从大体上有两种识别方式。

（1）自上而下的推导方式

自“上”而下语法分析是从文法的开始符号开始推导，直到最“下”层的句子，因此得名。从方法的开始符号往下推导，检查是否可以推导到实际的代码，如果可以，说明所输入的代码语法正确。比如一个树干延伸了多个树枝，每个树枝又长出多片树叶，推导便是从树干到树叶的过程。

（2）自下而上的归约方式

以最“下”层的源码串（即文法中的句子）往“上”层的非终结符“归约”，因此得名。归约

是推导的逆过程，推导是把非终结符（产生式的左部）按照产生式的右部“展开”为“终结符或其他非终结符”，归约是把“终结符或其他非终结符”按照同样产生式的右部“收敛”为非终结符（即产生式的左部）。归约是从最“下”层源码串为语法分析入口，检查是否可归约为某一产生式，产生式就是文法的规则，也就是判断代码是否符合语法，继续向上归约，直至开始符号。归约就是从树叶到树干的过程。

由于自上而下的推导方式比较容易用程序实现，大部分编译器都是采用此方式，这里就以它为主介绍语法分析。

自上而下推导就是检查从文法的开始符号（最大的非终结符）是否可推导出这个用户的代码，语法正确的话用户代码就是文法的句子，注意这里说的是句子，并不是句型，必须是把产生式中所有的非终结符推导到终结符为止。“语法正确的用户代码就是文法中的句子”这么说想想也很合理，你看，我们在给同事解释自己所负责模块的代码时，一般都会指着屏幕上的某行代码说：这“句”话的作用是啥啥啥。

匹配的过程就是非终结符推导的过程，如果源码语法正确，即源码是文法的一个句子的话，那么肯定能从文法的开始符号推导出用户的这句代码。举个例子，比如语法规则是前面所述的“文法 1”，如果句子是“小明踢球”，主语=>名词=>小明，谓语=>踢，宾语=>球，符合文法 1 中的规则。如果是源码是“小明和球”，这个“和”是连词，不符合文法 1 中的规则，因此就不符合语法，这就是语法分析的工作。当然，“山爬小明”虽然有语病，但还是符合语法规则的：主语是山，符合名词或代词，谓语是爬，符合动词，宾语是小明，符合名词或代码。估计有很多读者误以为判断句子是否有意义是语义分析的工作，这一定要纠正，这可不关语义分析的事儿。因为此类错误通常属于程序逻辑的问题，逻辑上的错误编译器可不好检查，比如某程序员误以为成年后就可以结婚了，而 18 岁就是成年，因此他在程序中判断男人必须大于等于 18 岁才可以结婚，但实际上我国婚姻法规定男人的法定结婚年龄是 22 岁，那么此程序员的这句大于等于 18 岁的代码的“意义”就错了，这种编程人员的逻辑错误难道也要让编译器检查出来吗，如果那样的话，编译器还得懂法律，那必须得是百科全书才行。显然这是不可能的。编译器制定了语法规则，只要用户遵守规则，怎么运作都行，如果不遵守规则，编译器就报错罢工，这已经是尽责了。

0.26 语法分析中的推导和归约为什么都要最“左”

其实这个很简单，只是编译原理比较难懂，而且很多读者都没实践过语法分析器，再加上编译原理书上很少有提到过原因（也许那些作者觉得很简单，没必要提），因此部分读者以为“最左”是规定。规定也是有理由的，肯定是为了规避某些问题而约定的，不能胡乱规定，通过下面的例子你也许自己能看出来为什么要最“左”。

自“上”而下语法分析是从文法的开始符号开始推导，通常它采用的是最左推导。最左推导是指每次都先推导最左边的非终结符，然后再推导右边的非终结符。比如如图 0-15 所示产生式。

```
S -> AB
A -> C
C -> c
B -> b
```

图 0-15

最左推导的过程如图 0-16 所示。

```
S -> AB
S => CB
S => cB
S => cb
```

图 0-16

自“下”而上的语法分析是从句子往上归约为某一产生式，通常是最左归约，即每次都先归约最左边的，被归约的可以是终结符、非终结符或两者的组合。比如用户源码是 cb，那么先将 c 按照产生式 C -> c 归约为非终结符 C，输入串就为 Cb，再将 C 按照产生式 A -> C 归约为非终结符 A，即 Ab。最左边的非终结符 c 处理完了，再处理右边的 b，根据规则 B -> b 归约为 B，即成为 AB，再根据规则 S -> AB，最终得到开始符号 S，说明语法正确。

两种语法分析都是最先处理“左”边的，介绍上面的两个例子下面说明下最“左”的理由，其实原因好理解，估计读者也想到了，就是因为用户的源码同写字的顺序一样都是从左往右写的，用户的代码逻辑肯定是从左往右读才能被正确理解，因此肯定要先处理最左边的字符才能正确理解用户的意思，因此最左边的字符要最先进来才对。

0.27 什么是语义分析

越是重要的东西就越是不容易察觉到其存在，当它成为我们必不可少的依赖时我们就容易忽略它，因为它已与我们融为一体，能感觉到的皆是身外之物。语义分析是整个编译过程中最重要的部分，以至于我们都不知道它在哪里。

语义分析混迹于语法分析和代码生成之中，功能并不像词法分析、语法分析那么独立、明显，因此不容易搞清楚它的作用。语义分析的工作可“不”是分析句子是否有意义、是否符合逻辑、通顺成句等等，语义分析的工作是识别出源码的功能逻辑，即搞清楚源码的意义、功能、意图是什么。搞清楚之后，就要按照这个意义生成目标代码了，因此语义分析是为“生成代码”服务的。这和翻译人员的工作是一样的，比如把中文翻译为英文，翻译人员听到中文后，自己先在脑子中体会、消化这段中文，然后按照他自己的“理解”翻译成对等意义的英文，这个“理解”中文的过程就是语义分析，你看，翻译人员这个“语义分析”没人看得到，人们只看到了他在两种语言之间切换输出。同样，编译的目的就是把源码编译为目标代码（或中间代码，本章有关此内容统称为目标代码），必须要搞清楚源码的意义后才知道怎么把它翻（编）译为对等意义的目标代码，因此它通常与语法制导一同工作（语法制导的任务就是翻译成目标代码），在理解源码之后马上输出对等的目标代码，因此人们只注意到了编译后的“实实在在”的目标代码，忽略掉了语义分析的存在，就像人们只注意到了翻译人员翻译后的英文一样，其实那个“理解”原文的过程才是整个翻译工作的重中之重。

既然语义分析是为生成代码服务的，因此很多书上都会把语义分析和生成代码放在一起介绍，为了更形象一点，看个例子。比如有一句这样的代码 $a=b+2$，进行语义分析后，知道了此行源码的意图是把 $b+2$ 的结果写入变量 a。然后开始为其生成目标代码，假如目标代码是汇编语言，那么就要生成：

```
mov tmp, 2
add tmp, b
mov a, tmp
```

即理解完了源码的语义后马上就生成对等的目标代码，这是一气呵成的。

好啦，本小节结束。

0.28 什么是语法制导

生成中间代码或目标代码不像记法分析和语法分析那样有法可依，可以很容易地量化到具体的实现，它没有固定的方法，因此这也是编译领域中较难的部分。而语法制导是生成中间代码或者目标代码的一种方式，仅是个概念，并不是具体的方法，它是指在语法分析的过程中同时完成绑定在产生式上语义规则的“动作”，这个动作便是指“生成代码”这个动作。也就是说，语法制导是指在语法分析的同时生成代码（中间代码或目标代码），从而完成编译。

编译的目的就是为了生成源程序对应的中间代码或目标代码，生成的代码肯定要能表达出源程序的功能逻辑才行，也就是说必须要如实表达出源程序的需求，因此生成代码的顺序要与源程序的功能逻辑的顺序相一致。如何保证逻辑一致？很简单，只要按照源程序的功能逻辑去分析代码就行了，这如何做到呢？还是很简单，因为语法肯定是不变的，按照语法去分析代码就行了，这个语法其实就是分析源码的“导航”，源码的功能逻辑肯定是遵循语法来实现的，因此按照“语法”去分析源码便识别出了源码的功能逻辑。一些传统的编译器会把此功能逻辑用树来记录，这个就是抽象语法树，即 AST，比如表达式 3+2 生成的语法树就是以+为根，以 3 和 2 分别为左、右子树的二叉树。也就是说，语法分析便是生成抽象语法树的过程，抽象语法树便是源码功能逻辑的本质表达，这就是“抽象”的意义所在——去掉不必要的部分，取其本质。当然，现在很少有编译器会生成抽象语法树了，尤其是对于一“遍”的编译器，原因是抽象语法树是在严格按照以语法为导向的分析过程中产生的，这个分析过程本身就是识别源码功能逻辑的过程，既然已经知道了源码的意图，那直接生成指令就行了，何必再额外生成个语法树呢，因此生成抽象语法树是完全没有必要的，即使对于某些多遍的编译器，抽象语法树也不是必要的，这一切都取决于实现。

啰唆之后你明白了，这个“导”是指“导向”，即生成的代码要以语法分析为导向，因此称为制导。

0.29 词法分析器吃的是 lex，挤出来的是 token

词法分析器的功能是将源码中的单词封装为 token，单词称为 lex（Lexeme），这里的“单词”不仅是英文单词，泛指文法中所能推导出的一切终结符，比如左小括号“(”也是单词，只不过是单个字符组成的单词。小节标题中的“吃”指的是输入，“挤”指的是输出。严谨地说，词法分析器“吃”的是长长的源码文本字符串，并不是 lex，因为 lex 是它从长长的源码串中解析出来的一小部分，这已是词法分析器处理文本之后的效果，因此并不是真正的输入。写成这样的标题是想让大伙儿注意到 lex 和 token 不是一回事儿，并且词法分析器在两者间起到承上启下的作用。

在最初的编译器中是没有词法分析器的，起初是由语法分析器代劳，其过程是：语法分析器从文件中获取单词，然后同已知的关键字比较，和哪个关键字匹配上了就表示该单词就是哪个关键字，否则就是用户自定义的标识符，这就要校验标识符的合法性，比如标识符不能以数字开头等等。同时还要记录该单词所在的行号，以备源码中若有语法错误就提示错误的地方是在哪行，甚至有时候需要知道上一个单词的相关信息等。你看，语法分析时需要“校验”单词和“收集”很多与单词相关的周边信息，而这个工作逻辑上其实是独立于语法分析的，那么语法分析器就干脆把这个工作拆分出来，这个拆出来的部分便是词法分析器。称为“器”之后显得有些“深不可测”了，其实在软件中这都是函数，只不过是把之前的工作封装为一个函数，语法分析器每次需要单词的相关信息时就调用这个函数（也就是词法分析器）。由于单词的相关信息还是挺多的，不

是一个简单的变量能存得下的，因此要用一个复合结构来存储，这个复合结构就是 token。

token 存在的另外一个原因是，它给了语法分析器一个“结果”，因此让语法分析更方便。比如若没有 token 的话，词法分析器把识别到的单词字符串（即 lex）返回给语法分析器，那么语法分析器如何知道这个单词是什么？肯定不知道，还是没有“结果”，还得和词法分析器一样再比较一次字符串，那么这个工作就重复了。

总结，词法分析器的工作就是，收集源码中各个单词的一系列周边信息，并把这些信息封装为 token 给语法分析器使用。

0.30 什么是“遍”

“遍”可以理解为编译器从头到尾扫描源码的次数，往往是次数越低编译器越高效，因此也能用来衡量编译器的性能。尽管有很多一遍的编译器，但多遍编译器也不少见，比如第一遍先收集所定义的全部变量，第二遍再判断所引用的变量是否已定义。或者为函数分配运行时栈的空间时，需要得知局部变量和参数的个数才能确定运行时栈的大小，那么第一遍先统计这些，第二遍时再确定分配栈空间的大小。

0.31 文法为什么可以变换

文法是语言的语法规则，因此可能给我们的“幻觉”是：“一种语言对应一种文法，文法变了语法规则就变了，就变成了另一门语言了，这不科学”。其实这是理解错误，应该是：一种文法对应一种语言，一种语言可以对应多种文法，如果多种文法描述的是同一种语言，那么就称这几种文法等价，文法变换就是把文法转换为另一种等价的文法，文法变换的基础就是文法等价。

举个例子，比如某个句子是奇数个字符 c，如 c、ccc、ccccc……它可以由文法

$$G1 = (\{S\}, \{c\}, \{S->cSc, S->c\}, S)$$

生成，也可以由文法

$$G2 = (S, A), \{c\}, \{S->cA, S->c, A->cS\}, S)$$

生成，也就是说文法 G1 和 G2 是等价的。现实生活中也有规则等价的例子，比如拿蒸馒头来说，和面时先放水还是先放面，蒸出的馒头都一样。

为什么要变换文法？这是为方便语法分析。因为语法分析就是按照语法规则去匹配用户的源码串，由于用户代码是自左向右写的，因此从左到右地处理用户代码串才是合理的，所以文法产生式右部中非终结符和终结符的位置顺序也应该符合源码从左到右的顺序。语法分析以产生式为导航，但是有时候会因语法规则本身设计的问题导致语法分析工作走入死胡同或者前进的方向不明确，这时候就需要变换一下规则以使语法分析可继续进行下去。有哪些情况会扰乱语法分析呢？请见“为什么消除左递归和提取左因子”小节。

0.32 为什么消除左递归和提取左因子

前面说过了，语法分析有两种方式，一种是自上而下的推导，另外一种是自下而上的归约。无论哪种方式，由于用户的代码是以从左到右的顺序书写的，只有从左到右地处理用户代码才能理解用户代码的逻辑，因此这两种语法分析方式都要优先处理用户源码中最左边的字符。那么语法分析是如何保证最左边的源码串先被处理呢，这就是产生式的作用了。我们说过，产生式就是语法分析的导航，确切地说是产生式右部中符号（非终结符和终结符）的位置顺序才是语法分析

的导航。这个顺序就是语法规则，语法的设计者知道用户习惯是以从左往右的顺序写字，他自己也不例外，写代码也是写字，因此也是按照从左到右的顺序来编制语法。语法是什么？就是一堆规则，规则是由产生式来体现的，确切地说是产生式右部，因此就按照从左到右的顺序来写产生式右部中的符号，这样就符合了用户源码的顺序。

用户得遵守此规则才能使用此语言，语法分析会读取这个规则，根据规则便知道用户源码串应该是什么，总之规则使源码串可预见，实现了最左边的源码串优先被处理。那么问题来了，产生式右部中最左边的符号可能是终结符，也可能是非终结符，如果是后者的话，该非终结符可能与产生式左部的非终结符同名，比如这种产生式：

$$P \overset{+}{\Rightarrow} Pa$$

右部中最“左”边的符号也是非终结符 P，在自上而下分析中，这样的推导会导致死循环，P 永远用 Pa 代替，无限“递归”下去，这就是左递归，即产生式右部中最左边的符号和产生式左部符号一样。最终推导的展开式为：

$$P \overset{+}{\Rightarrow} Paaaaaaaaaa\ldots\ldots$$

语法分析走不出来了。因此对于自上而下分析来说，必须要消除左递归。

消除左递归的方式通常是引用一新的非终结符，把左递归变成右递归，右递归就是产生式右部中最右边的符号和产生式左部符号一样。转换的公式我就不列出了，大伙儿有兴趣去查编译原理的书籍吧。

下面介绍提取左因子的必要性。假设非终结符用大写字母表示，我们知道，产生式的右部中可以包括多个以'|'分隔的候选式，比如这个表示 if 的非终结符的产生式：

```
IF_STAT -> if (EXP) {STAT} | if (EXP) {STAT} else {STAT}
```

这里说一下，自顶向下语法分析中，每个非终结符都对应一个函数，也称子例程，当源码串为 if(a > 0) {c++}时，语法分析器会执行非终结符 IF_STAT 的规则，即调用 IF_STAT 对应的子例程，该子例程应该选择哪个候选式呢？因为候选式前面的部分是一样的，都是 if (EXP) {STAT}，也就是说对于有公共前缀的候选式会让语法分析器失去方向很迷茫。因此为了更加明确地选择候选式，需要把左边的公共因子提取出来，就像数学中的提取公因式一样，此产生式改成：

```
IF_STAT -> if (EXP) {STAT} (ε| else {STAT})
```

其中的 ε 表示空，就是没有。或者用 EBNF 表示成：

```
IF_STAT -> if (EXP) {STAT} [else {STAT}]
```

其中的[]表示出现 0 次或 1 次，即有或没有。

话说回来了，如果各候选式没有公共前缀不就没这些问题了吗，如何判断是否有公共前缀呢？请见下节。

0.33 FIRST 集、FOLLOW 集、LL(1)文法

文法是人为设计出来的，为避免给语法分析制造麻烦，应该在设计之初就考虑规避一些问题。因此做文法设计的人最好做过语法分析，否则不容易清楚语法分析的需求。

之前说过提取左因子的原因是各候选式有公共的前缀，如果没有公共前缀就不用再提取啦，直接进行语法分析即可。如何确保没有公共前缀呢？你看，无论公共前缀多长，都是从第 1 个字符开始的，只要第 1 个字符不一样，就无法构造公共前缀啦。估计大伙儿都有答案了：只要保证产生式中各个候选式的首终结符的首字符不一样就一定没有公共前缀。假设产生式 A->a1|b2|c3，用公式表示则为：

$$FIRST(a1) \cap FIRST(b2) \cap FIRST(c3) = \varnothing$$

FIRST 表示终结符首字符集，上面的公式表示各候选式的 FIRST 集交集为空（∅表示空集）。注意，尽管这里的参数是终结符，但不要以为只能处理终结符，终结符首字符集是个集合，给 FIRST 的参数是什么，逻辑上它就返回相应终结符的首字符集。即如果参数是终结符，它就返回该终结符的首字符，如果参数是非终结符，它就自动进行推导，直到遇到终结符，然后返回终结符的首字符。切记，FIRST 求的是终结符首字符“集”，如 FIRST(a1)得到的只是该终结符的首字符 a，因此肯定不成“集”。FIRST 的参数还可以是非终结符，通过推导，该非终结符总会到达某一终结符，因此 FIRST 集最终还是求的终结符的首字符。比如 FIRST(A)={a,b,c}。

FIRST 集的提出是为避免用户源码可匹配产生式中“多个”候选式的一种辅助手段，只要各个候选式的 FIRST 集交集为空，语法分析器就不会面临如何选择的问题。倘若用户源码不属于任意一个候选式呢？难道就报语法错误吗？为时过早，在某种情况下，也许语法也是正确的。比如非终结符 A 的某个候选式为 ε（空），也就是说 A 可以用 ε 代替，要知道，非终结符也可以成为产生式的右部，在该右部中，碰巧非终结符 A 后面的终结符 b 可以匹配用户源码，而非终结符 A 恰好可以推导出 ε，也就是说相当于可以把非终结符 A 从产生式右部中去掉，那么终结符 b 就被提到原来非终结符 A 的位置，那么就正好满足用户代码的匹配，在这种情况语法就是正确的，这就引出了 FOLLOW 集。

FOLLOW 集称为后继终结符集，注意，并不是后继终结符首字符集。FOLLOW(A)是指出现在非终结符 A 之后的终结符或#，#表示输入结束符，类似文件结束符 EOF，表示源码都处理完了，#是为语法分析方便而人为加上的符号，产生式中并不存在。尽管 FOLLOW 集是为解决非终结符匹配空候选项后能用其后的终结符匹配用户源码而提出的，但这并不表示参数必须为非终结符，只是应用场景全是求非终结符后面的终结符，所以我们看到的参数才是非终结符。参数后面的终结符要么是推导出的，要么就是紧跟在参数后面的，如图 0-17 所示两个文法 FOLLOW(A)的结果都是{a}。

文法 1	S -> AB A -> ε B -> a
文法 2	S -> Aa A -> ε

图 0-17

另外，尽管前面说了非终结符 A 有个 ε 候选式，然后非终结符 A 后面的终结符与用户源码匹配，但这与 FOLLOW 集的定义无关，提出那个条件是为了引出下面 LL(1)文法的定义：

（1）文法中不含左递归；

（2）文法中所有非终结符的各候选式 a 的 FIRST(a)互不相交；

（3）文法中的每个非终结符 A，若它存在一个 ε 候选式，那么 FIRST(A)和 FOLLOW(A)交集为空。

满足以上 3 条的文法即是 LL(1)文法，其中的第一个 L 表示从左到右扫描源码串，第二个 L 表示最左推导（即每次推导产生式右部中最左边的非终结符），1 表示分析时的每一步只需要向前查看 1 个符号。提醒，“推导”就意味着自上而下，归约才意味着自下而上，后面会介绍。因此 LL(1)属于自上而下递归下降分析法，为什么是“下降”呢？因为在非终结符对应的子例程中，每次递归调用的都是该非终结符所属产生式的右部中的非终结符对应的子例程，相当于调用了自己的组成部分对应的子例程。

LL(1)是最简单易处理的文法，它是这样工作的。

（1）若输入串匹配到了非终结符的一个候选式，就执行该候选式对应的例程。

（2）若输入串不属于非终结符的任意一个候选式：若该非终结符存在一个 ε 候选式，且该非终结符的 FOLLOW 集匹配输入串，就让非终结符自动匹配 ε。否则就报语法错误。

0.34 最右推导、最左归约、句柄

既然是推导，就说明语法分析是自上而下，我们在前面介绍自上而下语法分析时都说了是最左推导，那么这个最右推导是干吗的？最右推导是为了解释最左归约。

前面说了，归约是推导的逆过程，而最左归约是最右推导的逆过程。走迷宫大伙儿都玩过吧，请教你个问题，你是怎么找到走出迷宫的路线的呢？有经验的你肯定是从出口往入口找，因为出口的路线较明确，就一条，不像在入口那样面对多条路，走哪一条路需要靠“撞大运”，发现走不通了再回来，这一现象在算法中称为回溯，即重新回到出发点再重新选路。回溯是最麻烦最讨厌的事情了，之前浪费的时间不说，回溯到之前的状态就意味着恢复到之前的环境，这个相对麻烦一些，因此减少回溯是提高效率的关键。那进一步讲，无回溯岂不是更好，如何保证无回溯？文法中的产生式有很多，在归约过程中也面临从众多路线中选路的问题，当有多个符合的产生式时该如何做出选择？假设源码的语法是正确的，如何确保源码能归约到开始符号以证明源码语法正确？思考一下迷宫的例子，从出口走到入口比较容易，容易确保迷宫走得通。我们知道一个句子可以从开始符号经过多步推导得到，因此该句子只要按照推导的逆序就能完美归约到开始符号啦，即怎么来的就怎么回去。之前说过了最左归约的原因，受迷宫启发，归约成功的前提是先保证从开始符号能推导成功，与最左归约相逆的便是最右推导。

顾名思义，最右推导就是在推导过程中每次都先推导最右边的非终结符，用户源码是从左到右写的，源码最后的部分肯定是最右边，因此最右推导的结果是用户源码中最先写入的部分（即源码最左边）是最后得到的，那么对于其逆过程来说就是最先处理的部分，也确实就应该这么做，前面说过了用户源码最左边的应该最先被处理，这才符合逻辑。最右推导的逆过程正是最左归约，完美契合。举个最右推导的例子，文法如图 0-18 所示。

```
S -> eAcBa
A -> Ab | b
B -> d
```

图 0-18

最右推导过程如图 0-19 所示。

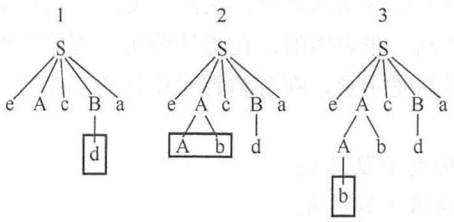


图 0-19

你看，以上 3 个推导过程中，都是根据某一产生式推导非终结符，产生式的右部我都用方框框起来了，它们是以非终结符为左部的产生式的“完整”右部，一定要记得“完整”这个特性。之前介绍过了短语的概念，方框中的就是短语，而且是直接短语。如在图 0-19 的 1 中，d 是句型

eAcda 相对于 B->d 的“直接”短语，它是由最“右”推导 B 时得到的。图 0-19 的 2 中，Ab 是句型 eAbcda 相对于 A->Ab 的“直接”短语，它是接上步的 eAcda 经最“右”推导 A 时得到的。当然也可以直接把 A 推导为 b，那就没后面的 3 什么事儿了。图 0-19 的 3 中，b 是句型 ebbcda（此时已经全是终结符，变成句子了，句子是特殊的句型）相对于 A->b 的“直接”短语，它也是上一步的 eAbcda 经最“右”推导 A 时得到的。把以上 3 个过程反过来看，各自方框里的短语都可以归约到上一步，即 3 变成 2，2 变成 1。因为方框中的短语是刚刚由上一步直接推导出来的，因此方框中的短语肯定能做到毫无差错地归约到上一步，这虽然像是废话，但却是最左归约的精髓，只有这样才能保证归约过程万无一失，零回溯。

我们应该都有过文件操作的经验，比如 open 一个文件后，系统返回的是这个文件的句柄。为什么叫句柄呢？因为它牵动着另一端更大的文件数据块，操作句柄就是操作一个文件。句柄在现实生活中就是“把手”（也叫柄），比如铁锅的把手，连着的是另一端的更大的锅，拿到了手柄就相当于获得了铁锅。总之，“柄”的特征就是自己虽然小，但它却“把持”着通往更多数据内容的入口。上图中方框里的是直接短语，尽管比较短小（因为短语就是句型的一部分），但它却是归约到上一句型的钥匙，好像是一个句型的“柄”，因此称方框中的部分为句柄。按照官方定义来说，句柄就是“最左直接短语”。你看，上图中的 3 个框框都是各自“最左”边的直接短语，符合定义。句柄就是最右推导中每一步的非终结符的直接展开项（即直接短语），最左边的肯定是最后展开的，但对于归约来说，正是应该最先归约的，因此最左边的就是最左归约的开始入口，最左归约中每次归约的就是句柄，句柄就是可归约串，只有归约的是句柄才能做到万无一失逆向回归到开始符号。

这里补充一下句柄容易弄混的部分，不是最左端的就一定是句柄。就拿上图中 1 最左边的“e”来说，它可不是句柄。句柄的概念包括“直接短语”“最左”，首先得是直接短语才行。直接短语是非终结符所在产生式的“完整”右部，如果有候选式的话得是“完整”的候选式，即必须是非终结符的完整展开项，那个“e”只是非终结符 S 的一部分，因此不是短语，更不是句柄了。

0.35 算符优先分析法

因为我们实践中采用的语法分析方法有一点点像算符优先分析法，所以本文对算符优先分析法的介绍只是蜻蜓点水，就当是一点点铺垫。

算符优先分析法也是一种自下而上语法分析，既然是自下而上，那语法分析的思路就是归约，但它并不是严格的最左归约，它是通过“算符”的优先级来寻找可归约串进行归约，因此称为“算符”优先分析法。注意！算符不仅包括运算符，它是广义的，泛指文法中的终结符。

算符的优先级是提前写入到一张表中的，在语法分析过程中读取这张表来比较不同算符的优先级。注：以下优先级的符号只是示意，在各自符号中间再加个‘.’才是文法中优先级比较符号。优先级比较结果有：

（1）$a > b$ 算符 a 的优先级大于算符 b；

（2）$a = b$ 算符 a 的优先级等于算符 b；

（3）$a < b$ 算符 a 的优先级小于算符 b。

注意，这里优先级和数学中的比较运算是不一样的，反过来不一定也成立。比如 $a > b$ 并不表示 $b < a$，原因是算符的大小不只和优先级本身有关，对于相同优先级的算符，先计算哪个是由结合性决定的，如右结合、左结合。因此算符的优先级是有序的，这个有序便指的是结合性，即算符的优先级还与出现的顺序有关。

0.36 算符优先文法

算符优先文法是用上节的算符优先分析法做语法分析的，在定义算符优先文法之前要先定义算符优先分析法。

在文法 G 中不存在形如 P->…AB…的产生式，其中 A 和 B 都是非终结符，就称 G 为算符文法。

为什么规定两个非终结符不能相继挨着呢？原因是算符肯定是终结符，而操作数可能是终结符或非终结符，如果两个非终结符相继挨着，展开后很可能是两个操作数相继挨着，这在数学运算中是不可能存在的，因为两个运算符之间只有 1 个操作数。相反，两个运算符却是可以相继，因为有的运算符是前缀，有的是后缀，相继挨在一起并不表示操作数是冲突的，比如 1- ++a。

假如 G 是一个不包含 ε 产生式的算符文法，对于做任意两个终结符 a 和 b。

（1）$a=b$ 当且仅当文法 G 中含有形如 P->…ab…或 P->…aXb…的产生式。

（2）$a<b$ 当且仅当文法 G 中含有形如 P->…aY…的产生式，且 $Y \overset{+}{\Rightarrow} b$…或 $Y \overset{+}{\Rightarrow} Xb$…。

（3）$a>b$ 当且仅当文法 G 中含有形如 P->…Yb…的产生式，且 $Y \overset{+}{\Rightarrow}$ …a 或 Y $\overset{+}{\Rightarrow}$ …aX。

如果文法 G 中任何终结符对儿（a,b）的优先级满足以上 3 种关系之一，就称文法 G 为算符优先文法。

对于 $a=b$ 还是没什么好说的，下面说说后两个。

高优先级体现在更加优先被处理，优先级越高，被处理的优先权就越高。在文法中先被推导的部分说明其优先级更高（因为更先被处理）。在第二项中，Y 是非终结符，其产生式包含终结符 b，在此 Y 先被推导处理，说明 b 的优先级更高，因此 $a<b$。第三项 $a>b$ 同理。不知你发现没有，将后两项中的 Y 展开后便是…ab…或…aXb…，同第一项一样。

0.37 非终结符中常常定义的因子和项是什么

在 BNF 中经常会定义非终结符 TERM 和 FACTOR，翻译过来就是项和因子，他们常用作表示表达式计算中的操作数。加减法的操作数通常称为项，优先级更高的乘除法的操作数称为因子。因此乘除法的操作数称为因子。

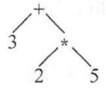
0.38 什么是抽象语法树

抽象语法树简称 AST（Abstract Syntax Tree），是对源代码的语法结构的抽象。抽象的意思是从事物中抽取本质性的特征，舍弃非本质的东西。因此抽象语法树就是源码的精髓。

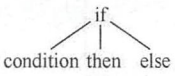
既然称为树，那么源码就是以树形结构来表示。树上的每个节点都表示源码中的一种结构。举个例子，表达式 3+2 若用抽象语法树来表示，是以+为根，3 和 2 分别为其左右子树的二叉树，如下所示。



稍微复杂一点的，3+2*5 的语法树如下所示。



语法树并不是都是二叉树，像 if-condition-then-else 这样的语句就是三叉树，如下所示。



前面也说过，很少有生成抽象语法树的编译器，因为语法分析的过程就是遍历抽象语法树，因此没必要单独再生成它了，我们也是。

0.39 编译器如何使用或实现文法中的产生式

文法是语言的总体语法规则，用产生式来细化到具体规则，规则越多产生式就越多。产生式毕竟只是规则，编译器如何去实现这个规则，很多读者都不是很清楚，因此在学习产生式时往往不知书上所云，其实很简单。

产生式是分为两大部分，左部是非终结符，右部是该非终结符对应的定义，也称展开项。产生式是语法规则，因此语法分析器中会去实现这个规则。为方便编码，一般都采用自上而下递归下降的语法分析，自上而下大伙儿都懂，就是从文法中最大的非终结符即开始符号不断推导，直到推导出全是终结符的句子。递归下降是什么？这就是我所说的编码方便的地方，要想利用有限的规则创造无限句子，文法的设计必须要灵活、允许包含“递归”定义，即定义中包含自己，这样才能以“不变”的语法规则“应”“万变”的代码句子。比如以下产生式（定义不完整，仅举例）：

```
STAT -> WHILE_STAT | IF_STAT | BLOCK
…略
WHILE_STAT -> 'while' '('EXPRESSION')' '{'STAT'}'
```

一般在语言中与定义无关的代码称为 statement，即语句，与声明、定义有关的代码称为 definition，即定义。非终结符 STAT 表示语句。非终结符 WHILE_STAT 定义 while 的语法规则，其产生式右部中包含了非终结符 STAT，也就是 while 的循环体代码是用 STAT 表示的。STAT 对应的规则中又包含了 WHILE_STAT，即表示支持这样的 while 嵌套。

while(expression) {while(expression){}}

既然文法是递归的，那么在实现中递归调用是免不了的（当然也可以用循环代替，但有一定难度），一种可行方案是把非终结符对应到一个函数（也称子例程），即一种规则用一个函数来解析，递归调用各种非终结符对应的函数就行了。递归调用也得有个调用入口，开始符号也是非终结符，它所对应的函数就是整个语法分析递归调用的入口，由于开始符号是最上层，往下逐层调用便是下降的过程，因此称为递归下降。比如可以把开始符号对应的函数定义为 program，在 program 中调用其他非终结符对应的函数，期间发生各种递归，直到读完整个代码，最终回到 program 函数，语法分析结束。

这些非终结符对应的函数做了什么呢？也很简单，就是按照该非终结符的产生式右部分析语法，假设关键字 while 是用来做循环，其产生式（语法规则）是 WHILE_STAT =>while'('循环条件表达式')''{循环体}'，WHILE_STAT 对应的处理例程是 parseWhile，其伪码如下：

```
parseWhile ( ) {
   match('(');    // 匹配 while 关键字后面的“(”，如果没有“(”就报语法错误:while 后面应该接“(”
   parseExp();    // 解析循环条件
   match(')');    // 匹配条件后面的“)”，如果没有“)”就报语法错误:循环条件后面应该接“)”
```

```
    match('{');    // 条件之后是代码体，匹配代码体前面的“{”
    stat();        // 解析循环体代码
    match('}');    // 匹配循环体后面的“}”
}
```

好啦，就介绍到此，后面的实践会让你更有体会。

0.40 程序计数器 pc 与 ip 的区别

这两个都是 CPU 中的概念。CPU 是执行指令的机器，它必须要知道待执行的指令在哪里，这正是程序计数器的作用。程序计数器也称 PC，即 Program Counter，用于记录下一条待执行指令的地址。由于 CPU 每次只执行一条指令，执行了多个指令后，就相当于给程序中的指令统“计”了“数”量，因此称为程序计数器。程序计数器只是个概念，在各个 CPU 中都有自己的实现，比如 ARM 体系的 CPU 存储下一条指令地址的寄存器就叫 PC，而 X86 体系 CPU 存储下一条指令的寄存器称为 IP，即 Instruction Pointer。

本章内容如果不清楚也不要紧，不影响第 1 章～第 10 章的阅读。

第1章　设计一种面向对象脚本语言

有没有感觉设计一门语言实在是太有意思了，可以自定义语法规则，我的“地盘听我的”。

1.1　脚本语言的功能

本书设计一门纯粹的面向对象脚本语言，任何语言都有个名词，这里给这个语言起个名字——sparrow（麻雀）。它支持的功能如下。

（1）变量

支持局部变量和局部变量的定义。

变量可引用、赋值。

内部复合数据类型以大写字符开头，如 System.print()

（2）基本数据类型

数值：包括整数和浮点数。

字符串：包括普通字符和 unicode。

list：列表，如 Python 中的 list。

支持字面量创建，如['a', 'b']和 new 方法创建。

元素通过下标 list'[索引]'获得。

map：哈希数组，如 Python 中的字典。支持字面量创建和 new 创建。字面量创建如

```
{
    'k1':v1
    "k2":v2
}
```

key 可以是任何数据类型。同样支持 new 方法创建。

value 通过下标 map'[key]'获得。

range：用以确定一段整数范围，用符号..表示。range 包括 from 和 to 两个成员，分别表示这段范围的起和始，用区间表示[from, to]，即包括 from 和 to。如 range“2..6”，2 就是 from，6 就是 to，“2..6”表示 2、3、4、5、6。“..”类似于 Python 中的分片操作符“:”，只不过我们包括了结尾的 to，而 Python 不包括，若用区间表示则 to 后面的是右小括号“)”。

（3）运算

数值：+、−、*、/、%。

逻辑：>、<、==、!=、||、&&、？:、|、&等。

位运算：>>、<<。

方法调用：.。

索引：[]。

字符串：+、%（字符串内的表达式）

（4）控制结构

支持 if-else 选择。

支持 while 循环。

支持 for 循环。

支持 break 退出循环。

支持 continue，跳过本次循环体后面的部分，继续下一轮循环。

支持 return 返回。

（5）函数

尽管这是一门面向对象语言，但也支持传统意义上的函数，用关键字 fun 实现函数定义。

函数也是用类实现。

支持函数重载。

（6）类

就是传统意义上的 class，包括类定义和类实例，静态类。

实现继承，所有类都是 object 类的子类。

类成员（也称域，或字段）必须先声明再引用。

方法包括 method、getter、setter、subscript、subscriptSetter 和构造函数。支持块参数，块参数的参数是用“||”括起来的参数列表，以逗号分隔。

支持静态方法。

（7）线程

支持线程创建及调度。

（8）模块

支持执行模块和模块内模块变量的单独导入。

（9）注释

行注释：//

块注释：/*　块注释　*/

以上列举若有遗漏则以实际代码为主。

1.2 关键字

有以下关键字被提前征用了。

var：用于变量定义。

fun：用于函数定义。

if：用于条件判断。

else：用于条件判断的 else 分支。

true：bool 值真。

false：bool 值假。

while：用于 while 循环。

for：用于 for 循环。

break：用于退出循环。

continue：用于结束本次循环并进入下一轮循环。

return：用于从函数返回。
null：空值。
class：用于类定义。
is：用于判断类是否为某类的子类，即“is a”。
static：用于设置静态文法。
this：用于指向本实例。
super：用于指向父类。
import：用于导入模块。

1.3 脚本的执行方式

我们采用传统的虚拟机作为执行方式，即要实现一个虚拟机。编译器先把源码编译为 opcode，再让虚拟机执行 opcode。

opcode 即操作码，是自定义的一套专供虚拟机执行的指令，后面我们在实现虚拟机时会详细介绍。

1.4 “纯手工”的开发环境

既然本着教学的目的我觉得应该拿出教学的诚意，因此这里所说的“纯手工”是指编码中不想借助 STL 或其他类似的泛型语言，没有第三方库，一切以最基础最原始的形式展现语言的奥秘，因此选择了 C 语言，确定地说是 C89，并不是较新的 C99 标准，诚意满满，让我们纯手工去编码吧。

基础开发环境是：

（1）宿主系统是 Linux，采用 CentOS release 6.8 (Final)；

（2）编译器是 gcc，版本是 gcc version 4.4.7 20120313 (Red Hat 4.4.7-17) (GCC)。

1.5 定义 sparrow 语言的文法

在第 0 章介绍的文法中，我们采用的是大写字母表示非终结符，小写字母表示终结符，然而我们也说过了，在现实中为了便于编程，一般都用正规文法来定义语言，正规文法说白了就是用正则表达式定义的文法，因此本小节的基础是正则表达式，为保证容易看懂，我会用最简单的方式书写正则表达式。

值得注意的是，正规文法与第 0 章中介绍的文法有很大的差别，主要是涉及终结符是用"表示，原因是正规文法中会涉及()、[]和{}，这些在正规文法中都是元字符，有其特殊含义：()表示成为一组，[]表示范围，{}表示重复。但在实际语言它们只是字符串字面量（即终结符），比如在语言中()表示函数名后面的括号，也可表示表达式中的小括号，[]表示下标索引，因此为避免冲突，正规文法中用单引号括起的是终结符。

其实语法和传统语言差不多，只是用文法来描述就显得生涩了。注意，正规文法中的[]与 EBNF 中的意义不同，在此表示范围，其中可以用-表示一段连续的范围，比如 0-9 就表示 0 至 9 之间（包括 0 和 9）的任何数；a-z 同理，表示字母 a 到字母 z 之间的任何字母。[]后面一般会接量词，当然量词不一定只用在[]之后，但它一定是用在某个字符之后以表示该字符的数量，其前不能没有字符。按照数量级别划分有 3 种量词，+表示重复出现 1 次以上，*表示重复出现 0 次以上，?表示



出现 0 次或 1 次，比如可用[\t]*表示 0 个或多个空白字符，其中\t 是 tab。注意此处[]中的空白是空格，为了突显这里有个空格就写了两个。.表示任意字符，包括控制字符比如回车等，|表示或者、任意其一，比如 a|b，表示 a 和 b 两者取其一，注意，|是对两边的整体有效，并不是只对紧邻|的有效。比如对于 ab|cd 的意思是 ab 或者 cd，如果想表达 abd 或 acd，可以用分组符号()，就是小括号对儿。()表示作为一组考虑，使相应的正则符号应用于整个组成员。

如果不了解正则，可以暂时先看说明中的解释。文法产生式请见表 1-1 所列。

表 1-1

非终结符	产生式	说明	
数字	num -> 0x[0-9a-f]+| 0[0-9]+| [0-9]+'.'? [0-9]+	支持 0x 开头的十六进制整数 支持 0 开头的八进制整数 支持十进制整数及浮点数	
字符串	string -> '"'[.]* '"'	用双引号“”括起来的 0 至多个字符是字符串	
变量名	id -> [a-zA-Z_]+ [a-zA-Z_0-9]*	变量名只允许以字符 a-z、A-Z 和_开头，后面接 a-z、A-Z、_和数字	
定义变量	varDef -> 'var' [\t]+ id('=' exp)?	一个 var 后面只支持一个变量声明 声明变量：var id 或初始化：var id = 表达式	

if-else 分支选择	ifStat -> 'if' '(' exp ')'
	(block|statements)
	('else' block)?
		if (表达式) {
		代码块
		} else {
		代码块
		}
			if (表达式)
			语句
			else
			语句

while 循环	whileStat -> 'while'
	'(' exp ')' (block|
	statements)
		while (表达式) {
		循环体代码块
		}
			while (表达式)
			语句

for 循环	forStat -> 'for'[\t]+
	id[\t]+'('exp')' (block|
	statements)
		for i (可迭代序列)
		{
		循环体
		}
			for i (可迭代序列)
			语句

		注意，循环变量 i 后面没有 in	
break	breakStat -> 'break'	关键字 break	
continue	continueStat -> 'continue'	关键字 continue	
return	returnStat -> 'return'[\t]+ exp	关键字 return 无结果空返回：return 返回表达式结果：return 表达式	
参数列表	paralist -> id?(,id)*	参数列表: 参数可有可无，或者有多个，以逗号分隔	

函数定义	funDef -> 'fun' id
	'('paralist ')' block
		fun 函数名(形参列表) {
		函数体代码
		}	

下标调用	subscriptCall -> id[\t]* '['exp']'	下标是指[]，通常用来索引数组元素（脚本语言一般不支持数组），列表元素和哈希数组元素 [\t]*表示 0 个以上的空格或制表符，即有没有空白符都行 如 list[0]、map[key]	

续表

非终结符	产生式	说明
getter 方法调用	`getterCall -> id[ \t]*` `block?`	id 表示方法名，调用时只可以传递块参数，块参数是指函数体，即传递一个函数作为参数。下同 这里省略了方法前的类“.”和实例“.”，在后面的非终结符 methodsCall 会补全。假设 count 是类中的 getter 文法，那么完整调用如实例对象.count
setter 方法调用	`setterCall -> id[ \t]*` `'='[ \t]*exp`	与 getter 比，setter 后面多了个=和表达式，由于 setter 只支持一个赋值参数，因此不支持块参数，后面会介绍到：块参数也算作一个参数。假设 age=为 setter 文法，完整调用是： 对象.age=22

method 调用	`methodCall -> id[ \t]*`
	`'('paraList')'[ \t]*block?`
		method 就是类中的函数，只不过在“调用”时支持块参数，如：
		对象.method(实参) {
		　块参数
		}
		强调!!!! 以上并不是定义，而是调用

类文法调用	`MethodsCall -> (id[ \t]*'.')?`
	`( methodCall |`
	`setterCall |`
	`getterCall )`
		(id[\t]*'.')?表示“实例对象名.”或“类名.”是可有可无的，这么做的原因是在类中调用本类中的文法不需要指定类名或实例对象。在类外调用方法的话必须要有“实例对象名.”或“类名.”

调用语句	`callStat -> subscriptCall |` `methodsCall`	调用包括下标调用和文法调用两类，如 list[1]是下标调用， instatant.method()是方法调用

语句列表	`statements -> (`
	`ifStat |`
	`whileStat |`
	`forStat|`
	`breakStat|`
	`continueStat|`
	`returnStat|`
	`callStat`
	`)*`
		参数可有可无，或者有多个，以逗号分隔

中缀运算符	`infixOp = '+' | '-' | '*' |`
	`'/' | '%' | '>' | '<' | '==' |`
	`'!=' | '>=' | '<=' | '&&' |`
	`'||' |'&' | '|' | '~' | '>>' |`
	`'<<'`
		中缀运算符即双目运算符

前缀运算符	`prefixOp = '-' | '!'`	
中缀运算符表达式	`infixExp -> exp` `([ \t*]infixOp [ \t]* exp)+`	中缀表达式，可以后接多个双目运算。非终结符 exp 在后面定义
前缀运算符表达式	`prefixExp -> prefixOp` `[ \t]*exp`	

表达式	`exp -> num |`
	`string |`
	`id |`
	`callStat |`
	`infixExp |`
	`prefixExp`
		递归调用表达式 exp

续表

非终结符	产生式	说明
语句块	block := '{' ('|' paralist '|')? statments '}'	语句块用大括号括起，支持块参数
实例域	instantField->varDef	实例域就是类的属性，属性各个对象自己的数据
静态域	staticField->'static'[\t]+ instantField	静态域就是类属性，属于所有类对象的共享数据
域	fieldDef -> instantField| staticField	
method 定义	methodDef -> 'static'?[\t]+ id'(' paralist')' block	

getter 定义	getterDef -> 'static'?[\t]+
	id [\t]+ block
		getter 是没有括号的文法，常用来返回类成员的值，逻辑上相当于只读，但功能不限制。如：
		count {
		return cnt
		}

setter 定义	setterDef -> 'static'?[\t]+id [\t]*'='[\t]*('id')' [\t]+ block	setter 是 getter 后面接一个=，常用来修改类成员的值，逻辑上相当于写，只支持一个赋值，但方法内功能不限制
类中方法定义	methodsDef -> (methodDef | getterDef | setterDef)*	

类定义	classDef -> 'class' id (< id)?
	'{'
	fieldDef
	methodsDef
	'}'
		<表示继承，比如类定义：
		class 类名 [< 基类] {
		var instantFieldA
		var instantFieldA
		static var staticField
		instantMethodA(){
		xxx
		…
		}
		static staticMethod(){
		xxx
		…
		}
		}

初次接触文法的读者可能对递归定义感到“消化不良”，比如非终结符 exp 是用于定义表达式，exp 是由 infixExp 等非终结符组成，而 infixExp 又是由 exp 组成，看上去有点死循环出不来了，但你不要忘了，infixExp 只是 exp 的其中一个组成部分，exp 还可以由 num、id 等指代，num 和 id 下面再无递归定义，这就是递归终止的条件。因此 infixExp 的组成部分 exp 也会是 num 或 id 等。

以上只是大体上定义了语法，并不全面，大家知道大概意思就行了。这种方法定义看上去比较抽象，下面是个具体的样本。提示：可能由于排版软件的问题导致代码显示很乱，大家可以看本书附带的源码，以后的其他代码也是如此，不再赘述。

```
[work@work sparrow]$ cat -n employee.sp
 1 class Employee {
 2     var name
 3     var gender
 4     var age
 5     var salary
 6     static var employeeNum = 0
 7     new(n, g, a, s) {
 8        name = n
 9        gender = g
10        age = a
11        salary = s
12        employeeNum = employeeNum + 1
13     }
14
15     sayHi() {
16        System.print("My name is " +
17              name + ", I am a " + gender +
18              ", " + age.toString + "years old")
19     }
20
21     salary {
22         return salary
23     }
24
25     static employeeNum {
26         return employeeNum
27     }
28 }
[work@work sparrow]$ cat -n manager.sp
 1 /*
 2     本文件中的代码只为演示语法，无任何意义，不用深究
 3     这里只演示部分功能，如果读者感兴趣，
 4     可以参考 core.c 中后面注册的原生方法和 core.script.inc 中的脚本方法
 5     测试有限，难免还会有 bug，读过本书后应该有 bugfix 的能力，看好你，兄弟！
 6                                                 刚子
 7                                                 2018.4.12
 8 */
 9
10 import employee for Employee
11 var xh =  Employee.new("xiaohong", "female", 20, 6000)
12 System.print(xh.salary)
13
14 var xm =  Employee.new("xiaoming", "male", 23, 8000)
15 System.print(xm.salary)
16
17 System.print(Employee.employeeNum)
18
19 class Manager < Employee {
20     var bonus
```




```
    bonus=(v) {
        bonus = v
    }

    new(n, g, a, s, b) {
        super(n, g, a, s)
        bonus = b
    }

    salary {
        return super.salary + bonus
    }

}

fun employeeInfo() {
    System.print("number of employee:" + Employee.employeeNum.toString)
    var employeeTitle = Map.new()
    employeeTitle["xh"] = "rd"
    employeeTitle["xm"] = "op"
    employeeTitle["lw"] = "manager"
    employeeTitle["lz"] = "pm"

    for k (employeeTitle.keys) {
        System.print(k + " -> " + employeeTitle[k])
    }

    var employeeHeight = {
        "xh": 170,
        "xm": 172,
        "lw": 168,
        "lz": 173
    }
    var totalHeight = 0
    for v (employeeHeight.values) {
        totalHeight = totalHeight + v
    }
    System.print("averageHeight: %(totalHeight / employeeHeight.count)")

    var allEmployee = ["xh", "xm", "lw", "lz"]
    for e (allEmployee) {
        System.print(e)
    }

    allEmployee.add("xl")
    System.print("all employee are:%(allEmployee.toString)")
    var idx = 0
    var count = allEmployee.count
    while (idx < count) {
        System.print(allEmployee[idx])
```

```
        idx = idx + 1
    }

    System.gc()  //可以手动回收内存

    var a = 3 + 5 > 9 - 3  ? "yes" : "no"
    if (a.endsWith("s")) {
        System.print(System.clock)
    } else {
        System.print("error!!!!!")
    }

    var str = "hello, world."
    System.print(str[-1..0])
}

var lw = Manager.new("laowang", "male", 35, 13000, 2000)
System.print(lw.salary)
lw.bonus=3100
System.print(lw.salary)
var lz = Manager.new("laozheng", "male", 36, 15000, 2300)
System.print(lz.salary)

var thread = Thread.new(employeeInfo)
thread.call()
```

以下是上述文件的执行结果，其中的 spr 是最终的脚本解释器（包括编译器及虚拟机），spr 是 sparrow 的缩写。

```
[work@work sparrow]$ ./spr manager.sp
6000
8000
2
15000
16100
17300
number of employee:4
xm -> op
lz -> pm
lw -> manager
xh -> rd
averageHeight: 170.75
xh
xm
lw
lz
all employee are:[xh,xm,lw,lz,xl]
xh
xm
lw
```




```
lz
xl
1468814885
.dlrow ,olleh
[work@work sparrow]$
```

以上./spr manager.sp 就是执行脚本文件 manager.sp，这与任何脚本语言的运行方法都是一致的，执行过就是脚本的输出，大家有兴趣可以核对一下结果，除了 System.clock 返回的时间戳是动态变化的外，其他不变。

第2章　实现词法分析器

本章实现编译器的最前端——parser，即词法分析器。个人觉得词法分析器是编译器中最容易实现的部分，因此本章会很轻松。

代码风格如下。

缩进：3 个空格。

标识符：小驼峰式。

复合数据结构：首字母大写。

宏：下划线命名法 + 大写。

2.1 柔性数组

柔性数组（flexible array）是 C99 中的玩意儿，在 C89 中并没有。起初柔性数组叫柔性数组成员（flexible array member），因为它必须要放在一个结构体中作为结构体成员存在，下面介绍下它的来头。

看似功能简单的语言，往往“更底层”且“更灵活”，受限制较少，在自由发挥之下往往可碾压那些高级语言，比如汇编和 C 就是创建力无限的两种语言。就像盖房子，底层语言给用户提供的是建筑材料，比较累但是想盖啥样的房子都行，高级语言提供的是房子的结构，只需要小部分工作房子就成型了，因此盖房子更快但是风格单一。就拿 C 语言来说，它可是开发 linux 内核的语言（不过 linus 说没有采用 C++的原因是好的 C++程序员很少），C 程序员才是对代码精练的极致追求者，这也正是柔性数组诞生的起因：在结构体中经常需要单独存储一个字符串，由于字符串长度不固定，一种较常见的做法是在结构体中加一个指针作为结构体成员，用该指针指向字符串。举个例子，假设结构体是从堆中动态分配的，这和咱们编译器中的应用更接近，见下例。

```
1 struct sample {
2    int length;
3    char* str;
4} ;
5 struct sample* s = malloc(sizeof(struct sample));
6 s->length = 10;
7 s->str = malloc(s.length+1);
8 strcpy(s->str, "abcdefghij", 10);
9 s->str[s->length] = '\0'
```

功能实现了，此方法完全可以胜任。但是这样一来，字符串 str 占用的内存和结构体 s 本身是分离的，而且字符串需要额外维护一个用 malloc 申请的动态内存，因此 C 程序员认为还有可优化的地方——如果让字符串本身就位于 struct 中不是更好吗？你也想到了，在 struct 的末尾放一个数



仅供非商业用途或交流学习使用

组就行了，此数组专门用来存储字符串，但是我们是想让结构体能存储不同长度的字符串，可是字符串长度若不固定的话，结构体的长度也就不固定了，编译器需要在编译时就确定出数据类型的主要目的就是确定该数据类型的长度，即内存大小，因此不同的长度就等同于定义了多种不同的数据类型（或结构），我们不能为每种长度的字符串单独建一个 struct 吧，因此为使结构体长度固定，位于结构体中的数组的长度应该为 0。把结构体改为以下代码。

```
1   struct sample {
2     int length;
3     char str[0];
4   } ;
5   char* string = "abcdefghij";
6   int length = strlen(string);
7   struct sample* s = malloc(sizeof(struct sample) + length + 1);
8   s->length = length + 1;
9   strcpy(s->str, string, length);
10  s->str[s->length] = '\0'
```

见第 3 行，尽管数组 str 的长度为 0，但后面第 8～10 行我们把它当成包括实际字符串长度的数组来运用，这虽会涉及越界，但只要做好边界控制就可以。其内存布局如图 2-1 所示。

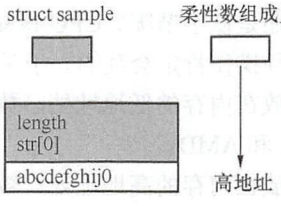


图 2-1

其中灰色的部分便是结构体 struct sample 原本的大小。

但是 C 标准规定不能定义 0 长度的数组，所以，有些编译器就把 0 长度的数组作为自己的非标准扩展，毕竟得民心者得天下，GCC 就支持此类型数组。鉴于标准是来自于用户的合理需求（忽视用户需求的标准必然也会被用户忽视），较新的 C 标准 C99 收录了此标准，它使用不完整类型来实现了此需求，因此柔性数组成员被“扶正”了。C99 规定，柔性数组成员在结构中只以符号地址的身份存在，并且一定是结构体的最后一个成员。以上结构体在 C99 中如下所示。

```
1   struct sample {
2     int length;
3     char str[];
4   } ;
```

str 数组名后的[]中并没有元素个数，因此 str 只作为符号地址存在于结构体中。提醒一下，尽管这里是用字符举例，但任何类型都可以用于柔性数组成员。

好啦，收工。

2.2 什么是字节序

声明一下，本小节部分内容是从《操作系统真象还原》第 0.19 小节摘录的，在此并不是为了宣传我那本书，我是怕读过那本书的读者会说我懒惰，毕竟多数情况下自己先主动承认错误挨的

仅供非商业用途或交流学习使用

骂也会少一些。

本系统中很多地方都用大端字节序来写数据，因此单独介绍下字节序。先说下为什么会产生字节序的问题。

内存是以字节为单位读写的，其最小的读写单位就是字节。故如果在内存中只写入一个字节，一个内存的存储单元便可将其容纳了，只要访问这一内存地址就能够完整取出这 1 字节。可是 1 字节要能够表示的范围只有 0～255（先只考虑无符号数），超过这个范围的数，只好用多个字节连在一起来表示。因此，在我们的 32 位程序中，定义的数据类型很多。1 字节的数据类型只有 char 型，像 int 型要占 4 字节，double 型要占用 8 字节。正如解决了一个问题又抛出了新的问题一样，解决了数值范围的问题，那带来的新的问题是，这多个字节该以怎样的顺序排放呢？一个超过 255 的数字必然要占用 2 个字节以上，这两个字节，在物理内存中，哪个在前？哪个在后？拿 0x1234 举例，数值中的高位 12 是放在内存的高地址处还是低地址处？

为什么要强调这个顺序呢？因为这涉及 CPU 得知道先获得的部分数据是属于高字节还是低字节，原因是读取内存数据的动作既是“随机”又是“线性”，随机是指内存是个随机存取设备，可以跳到内存任意位置去读写数据，线性是指内存读取是从低地址往高地址方向，像根线一样，假如给出了某变量的内存地址后，数据寻址操作是以该地址为最低地址，顺着此地址往高地址方向读取一定偏移量的数据，此偏移量正是该变量的数据类型占用的字节数，这样才获得了数据。既然内存读取是从低地址往高地址方向线性发展，对于 2 字节以上的数据，那么从低处往高处读的过程中，先遇到的是数据的高字节还是低字节呢？CPU 得知道此时获得的是数据的哪个部分，否则连这个都分不清的话，像移位这种操作肯定会乱的。于是就产生了这两种相反的排列顺序。

（1）小端字节序是数值的低字节放在内存的低地址处，数值的高字节放在内存的高地址。典型代表是 x86 架构的 CPU，如 INTEL 和 AMD。

（2）大端字节序是数值的低字节放在内存的高地址处，数值的高字节放在内存的低地址。典型代表是 IBM、Sun。

总结，从低地址往高地址读写内存的过程中，先遇到的是数据的哪部分就是哪种字节序，即先遇到的是高字节就是大端字节序，先遇到的是低字节就是小端字节序。

字节序只是规定了数据的高低字节的写入顺序，尽管字节序最初源自于 CPU“硬件”的需求，但后来网络传输、编码、同步数据等场景也遇到了 2 字节以上的数据类型拆分传输的问题，所以字节序的应用场景并不专指 CPU，“软件”中读写超过 1 字节的数据类型时也会涉及字节序的概念。

2.3 一些基础的数据结构（本节源码 stepByStep/c2/a）

在编写具体的词法分析器之前有一些前期的准备工作。一些通用工具类的代码会放在 include 目录下，请见 common.h 文件。

stepByStep/c2/a/include/common.h

```
1 #ifndef _INCLUDE_COMMON_H
2 #define _INCLUDE_COMMON_H
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <stdint.h>
6
7 typedef struct vm VM;
8 typedef struct parser Parser;
```

仅供非商业用途或交流学习使用


```
 9 typedef struct class Class;
10
11 #define bool char
12 #define true   1
13 #define false  0
14 #define UNUSED __attribute__ ((unused))
15
16 #ifdef DEBUG
17     #define ASSERT(condition, errMsg) \
18         do {\
19             if (!(condition)) {\
20                 fprintf(stderr, "ASSERT failed! %s:%d In function %s(): %s\n", \
21                   __FILE__, __LINE__, __func__, errMsg); \
22                 abort();\
23             }\
24         } while (0);
25 #else
26     #define ASSERT(condition, errMsg) ((void)0)
27 #endif
28
29 #define NOT_REACHED()\
30     do {\
31         fprintf(stderr, "NOT_REACHED: %s:%d In function %s()\n", \
32             __FILE__, __LINE__, __func__);\
33         while (1);\
34     } while (0);
35
36 #endif
```

第 7～9 行用 typedef 定义的 VM、Parser 和 Class 的完整实现分别在后面章节的文件中，前期有很多文件都需要这 3 个结构的定义，而且往往与很多其他结构相互依赖，即 A 包含 B，B 又包含 A，因此先包含这个文件以实现前向引用，此时不用过多关注。

如果函数中的一部分形参并未使用（后面你会看到，我们在系统中使用了一些函数指针来指向“通用”的函数声明，“通用”只是适合大部分情况，有些函数调用并不需要提供那么多实参），编译时会发出警告，这种情况下可以用 gcc 内部的属性__attribute__ ((unused))告诉 gcc 不要报警，为方便使用，在第 14 行定义宏 UNUSED。

为方便系统调试，系统中肯定要有很多断言，即 ASSERT，但 ASSERT 只应该出现在调试阶段，如果不在调试阶段，ASSERT 代码应该去掉，这有两个好处。一是代码更加简洁，程序体积会更小。二是减少了无用的代码，程序速度会更快。

通过修改源码去掉 ASSERT 肯定太麻烦了，即使是用替换工具也冒着错改源码的风险，既然是宏，那么最好是在预处理阶段搞定。第 15～27 行定义的 ASSERT 根据是否定义了宏 DEBUG 做出判断，宏 DEBUG 是我们定义在 makefile 中的，用于定义调试的开关，后面有一小节专门讲了这个 makefile，里面介绍了宏 DEBUG 的定义方法。回来继续，如果定义了宏 DEBUG，ASSERT 就是第 17～24 行的代码。否则的话就为第 26 行的((void)0)，达到了去掉 ASSERT 的目的。

第 29 行的宏 NOT_REACHED 表示“不可能到达”，通常有一些程序分支是不可能被执行到的，如果被执行到，说明程序出错。为检测这种情况定义此宏，将其安插到那些不可能到达的分支中。

仅供非商业用途或交流学习使用

下面看另一个基础文件 utils.c 的实现。

stepByStep/c2/a/include/utils.c

```
#include "utils.h"
#include "vm.h"
#include "parser.h"
#include <stdlib.h>
#include <stdarg.h>

//内存管理 3 种功能:
//    1 申请内存
//    2 修改空间大小
//    3 释放内存
void* memManager(VM* vm, void* ptr, uint32_t oldSize, uint32_t newSize) {
    //累计系统分配的总内存
    vm->allocatedBytes += newSize - oldSize;

    //避免 realloc(NULL, 0)定义的新地址，此地址不能被释放
    if (newSize == 0) {
        free(ptr);
        return NULL;
    }

    return realloc(ptr, newSize);
}

// 找出大于等于 v 最近的 2 次幂
uint32_t ceilToPowerOf2(uint32_t v) {
    v += (v == 0);  //修复当 v 等于 0 时结果为 0 的边界情况
    v--;
    v |= v >> 1;
    v |= v >> 2;
    v |= v >> 4;
    v |= v >> 8;
    v |= v >> 16;
    v++;
    return v;
}

DEFINE_BUFFER_METHOD(String)
DEFINE_BUFFER_METHOD(Int)
DEFINE_BUFFER_METHOD(Char)
DEFINE_BUFFER_METHOD(Byte)

void symbolTableClear(VM* vm, SymbolTable* buffer) {
    uint32_t idx = 0;
    while (idx < buffer->count) {
       memManager(vm, buffer->datas[idx++].str, 0, 0);
    }
    StringBufferClear(vm, buffer);
```




仅供非商业用途或交流学习使用


```
48 }
49
50 //通用报错函数
51 void errorReport(void* parser,
52         ErrorType errorType, const char* fmt, ...) {
53     char buffer[DEFAULT_BUFFER_SIZE] = {'\0'};
54     va_list ap;
55     va_start(ap, fmt);
56     vsnprintf(buffer, DEFAULT_BUFFER_SIZE, fmt, ap);
57     va_end(ap);
58
59     switch (errorType) {
60        case ERROR_IO:
61        case ERROR_MEM:
62           fprintf(stderr, "%s:%d In function %s():%s\n",
63                   __FILE__, __LINE__, __func__, buffer);
64           break;
65        case ERROR_LEX:
66        case ERROR_COMPILE:
67           ASSERT(parser != NULL, "parser is null!");
68           fprintf(stderr, "%s:%d \"%s\"\n", ((Parser*)parser)->file,
69                   ((Parser*)parser)->preToken.lineNo, buffer);
70           break;
71        case ERROR_RUNTIME:
72              fprintf(stderr, "%s\n", buffer);
73           break;
74        default:
75           NOT_REACHED();
76     }
77     exit(1);
78 }
```

函数 realloc 的声明是 void *realloc(void *ptr, size_t size)，表面上看其功能是将 ptr 指向的内存调整为 size 大小，但此函数实际上是具备 3 个功能。

（1）若 ptr 为 NULL 且 size 不为 0，功能等于 malloc(size)，用于申请内存。

（2）若 ptr 不为 NULL 且 size 为 0，功能等于 free(ptr)，用于释放内存。

（3）若 ptr 和 size 都不为 0，用于扩容。扩容有可能是原地扩容或者是重新分配一块内存，对于前者，是在原内存空间的延长线上继续分配，因此返回的内存地址还是之前的地址。对于后者，会重新分配一块内存以满足 size 大小的需要，此时功能相当于 free(ptr)和 malloc(size)的组合。

这里多说一句，对于第 3 种功能如果使用不当会造成内存泄漏。原因是，当 realloc 申请内存失败（比如内存不足）则会返回 NULL，因此 realloc 的调用方一定要妥善处理好返回值，最好不要把 realloc 的返回值直接赋给原指针变量，避免当 realloc 失败时原指针变量被 NULL 替换，从而原地址空间丢失，无法释放从而产生内存泄漏。

函数 memManager 是本系统的内存管理器，负责内存的分配、扩容和释放，它的核心就是用 realloc 来实现 malloc、realloc 和 free 3 种功能，包括 4 个参数：vm 是虚拟机指针，待以后介绍虚拟机时会和大伙儿细说，ptr 是由 malloc 返回的地址或者是 NULL，oldSize 是原来申请的内存量，



仅供非商业用途或交流学习使用

newSize 是变更后的内存量。包括 3 种功能。

（1）申请内存（等同于 malloc）。

参数要求：ptr 为 NULL，oldSize 为 0，newSize 不为 0，返回分配的地址。

（2）修改由 malloc 分配的空间大小，类似 realloc。

参数要求：ptr 不为 NULL，oldSize 为之前分配的内存大小，newSize 为修改后的内存大小。

（3）释放内存（等同于 free）。

参数要求：ptr 不为 NULL，newSize 为 0。

参数 vm 的用意主要体现在第 13 行，将系统申请的内存总量统计在 vm->allocatedBytes 下，将来会用它作为 gc 的触发条件。

再说一下第 16 行代码的用意，当函数参数 ptr 不为 NULL 且 newSize 为 0 时，memManager 的作用相当于 free(ptr)，这没问题。但是如果 ptr 为 NULL 且 newSize 为 0 时，会出现什么现象呢？可见下面代码。

```
[work@work tmp]$ cat -n reallocTest.c
     1 #include <stdlib.h>
     2 #include <stdio.h>
     3 void main() {
     4     printf("%p\n", realloc(NULL, 0));
     5 }
[work@work tmp]$ gcc reallocTest.c
[work@work tmp]$ ./a.out
0x25c1010
[work@work tmp]$
```

你看，realloc(NULL,0)除了具有 free 的作用外，还会返回一个非空的指针，相当于定义了一个新指针，而我们本来是想除了回收内存空间外也要回收指针的。为了防止这种行为，干脆对于 newSize 为 0 的所有情况（无论 ptr 是否为 NULL）都统一用 free 函数来单独处理。

回头接着看 utils.c 中第 25 行的 ceilToPowerOf2，此函数功能是找出大于等于 v 最近的 2 次幂，比如参数 v 为 3 时会返回 4，参数 v 为 12 时会返回 16，有点类似“进一法取整”，但是如果参数 v 已经是 2 的幂时返回的就是参数啦。本函数就是公式 1U<<(log(v−1)+1)的实现，粗糙的原理就是向下找出离 v 最近的 2 的幂，然后将该数再乘一个 2，即多一次乘方。但 v 有可能已经是 2 的幂，因此将 v 减 1 后再开方就是最保险的做法，将开方的结果加 1，这就是最终的 2 的次数，假设次数为 n，将此 n 作为左移的位数，就实现了连续乘以 n 个 2，也就是实现了 2 的 n 次方。公式 1U<<(log(v-1)+1)的步骤可以分为两步：先开方，再乘方。为了提速（减少系统调用和函数调用），函数的实现是用位操作完成的，它虽然在不断地做位移，但相当于是一步完成的，因为它在通过右移也就是开方的过程中，把开方的结果与原来的 v 值通过“位或赋值运算符|=”不断合并，保证了新的累计值 v 包含了相应的幂，因此不需要再单独乘方了。参数 v 的类型是 uint32_t，即 32 位整数，v 右移的位数分别是 1、2、4、8、16，它们加起来的和是 31，即对 v 开 31 次方，这已经满足了 32 位整数的开方。此算法来自于斯坦福大学。

接下来的宏 DEFINE_BUFFER_METHOD 和函数 symbolTableClear 是在 utils.h 中定义的，一会再说。

errorReport 函数是通用的报错函数，支持多种报错。ERROR_IO 是有关文件 IO 的错误，ERROR_MEM 是有关内存分配的错误，ERROR_LEX 是有关词法分析的错误，ERROR_COMPILE 是有关编译的错误，ERROR_RUNTIME 是有关运行时的错误，即在虚拟机运行阶段的错误。

下面是 utils.h 文件。

仅供非商业用途或交流学习使用

stepByStep/c2/a/include/utils.h

```
#ifndef _INCLUDE_UTILS_H
#define _INCLUDE_UTILS_H
#include "common.h"

void* memManager(VM* vm, void* ptr, uint32_t oldSize, uint32_t newSize);

#define ALLOCATE(vmPtr, type) \
    (type*)memManager(vmPtr, NULL, 0, sizeof(type))

#define ALLOCATE_EXTRA(vmPtr, mainType, extraSize) \
        (mainType*)memManager(vmPtr, NULL, 0, sizeof(mainType) + extraSize)

#define ALLOCATE_ARRAY(vmPtr, type, count) \
    (type*)memManager(vmPtr, NULL, 0, sizeof(type) * count)

#define DEALLOCATE_ARRAY(vmPtr, arrayPtr, count) \
    memManager(vmPtr, arrayPtr, sizeof(arrayPtr[0]) * count, 0)

#define DEALLOCATE(vmPtr, memPtr) memManager(vmPtr, memPtr, 0, 0)

uint32_t ceilToPowerOf2(uint32_t v);

typedef struct {
    char* str;
    uint32_t length;
} String;

typedef struct {
    uint32_t length; //除结束\0之外的字符个数
    char start[0];   //类似c99中的柔性数组
} CharValue;         //字符串缓冲区

//声明buffer类型
#define DECLARE_BUFFER_TYPE(type)\
    typedef struct {\
        /* 数据缓冲区 */    \
        type* datas;\
        /*缓冲区中已使用的元素个数*/\
        uint32_t count;\
        /*缓冲区容量用*/\
        uint32_t capacity;\
    } type##Buffer;\
    void type##BufferInit(type##Buffer* buf);\
    void type##BufferFillWrite(VM* vm, \
            type##Buffer* buf, type data, uint32_t fillCount);\
    void type##BufferAdd(VM* vm, type##Buffer* buf, type data);\
    void type##BufferClear(VM* vm, type##Buffer* buf);

```

仅供非商业用途或交流学习使用


```
49 //定义 buffer 方法
50 #define DEFINE_BUFFER_METHOD(type)\
51    void type##BufferInit(type##Buffer* buf) {\
52       buf->datas = NULL;\
53       buf->count = buf->capacity = 0;\
54    }\
55 \
56    void type##BufferFillWrite(VM* vm, \
57          type##Buffer* buf, type data, uint32_t fillCount) {\
58       uint32_t newCounts = buf->count + fillCount;\
59       if (newCounts > buf->capacity) {\
60          size_t oldSize = buf->capacity * sizeof(type);\
61          buf->capacity = ceilToPowerOf2(newCounts);\
62          size_t newSize = buf->capacity * sizeof(type);\
63          ASSERT(newSize > oldSize, "faint...memory allocate!");\
64            buf->datas = (type*)memManager(vm, buf->datas, oldSize, newSize);\
65       }\
66       uint32_t cnt = 0;\
67       while (cnt < fillCount) {\
68          buf->datas[buf->count++] = data;\
69          cnt++;\
70       }\
71    }\
72 \
73    void type##BufferAdd(VM* vm, type##Buffer* buf, type data) {\
74       type##BufferFillWrite(vm, buf, data, 1);\
75    }\
76 \
77    void type##BufferClear(VM* vm, type##Buffer* buf) {\
78       size_t oldSize = buf->capacity * sizeof(buf->datas[0]);\
79       memManager(vm, buf->datas, oldSize, 0);\
80       type##BufferInit(buf);\
81    }
82
83 DECLARE_BUFFER_TYPE(String)
84 #define SymbolTable StringBuffer
85 typedef uint8_t Byte;
86 typedef char Char;
87 typedef int Int;
88 DECLARE_BUFFER_TYPE(Int)
89 DECLARE_BUFFER_TYPE(Char)
90 DECLARE_BUFFER_TYPE(Byte)
91
92 typedef enum {
93     ERROR_IO,
94     ERROR_MEM,
95     ERROR_LEX,
96     ERROR_COMPILE,
97     ERROR_RUNTIME
98 } ErrorType;
```




仅供非商业用途或交流学习使用


```
 99
100 void errorReport(void* parser,
101         ErrorType errorType, const char* fmt, ...);
102
103 void symbolTableClear(VM*, SymbolTable* buffer);
104
105 #define IO_ERROR(...)\
106     errorReport(NULL, ERROR_IO, __VA_ARGS__)
107
108 #define MEM_ERROR(...)\
109     errorReport(NULL, ERROR_MEM, __VA_ARGS__)
110
111 #define LEX_ERROR(parser, ...)\
112     errorReport(parser, ERROR_LEX, __VA_ARGS__)
113
114 #define COMPILE_ERROR(parser, ...)\
115     errorReport(parser, ERROR_COMPILE, __VA_ARGS__)
116
117 #define RUN_ERROR(...)\
118     errorReport(NULL, ERROR_RUNTIME, __VA_ARGS__)
119
120 #define DEFAULT_BUFFER_SIZE 512
121
122 #endif
```

第 7 行定义的宏 ALLOCATE 的功能是分配类型为 type 的内存块，调用的是之前介绍的 memManager，根据 memManager 3 种功能的参数要求填入合适的参数，此处是申请内存，原指针 ptr 为 NULL，原内存大小 oldSize 为 0，新内存大小 newSize 为 sizeof(type)。提醒一下，如果是为原内存扩容，ptr 肯定不为 NULL，oldSize“通常”不为 0。ptr 不为 NULL 这很好理解，毕竟扩容是指改变原来内存的大小（缩小也是扩容，就称为负扩容吧，类似经济负增长），那总得提供下原来内存的指针。oldSize“通常”不为 0 只是逻辑上的，逻辑上扩容不可能是从 0 到 newSize，否则那是申请内存。但实际上内存扩容主要取决于 newSize，因此 oldSize 即使为 0 也没关系，提供 oldSize 的目的仅是为了统计内存使用量：用 newSize 与 oldSize 的差作为内存增量附加到 vm->allocatedBytes（可复习下前面 utils.c 中的第 13 行），因此在扩容的情况下 oldSize 肯定不为 0，非要写成 0 只会造成内存统计错误。

第 10 行的宏 ALLOCATE_EXTRA 用于分配除主类型 mainType 外还需要额外 extraSize 大小的内存，用于实现柔性数组的功能，此处如果你不是很清楚也没关系，在后面实践中还是很简单的。

第 13 行的宏 ALLOCATE_ARRAY 用于分配数组内存。

第 16 行的宏 DEALLOCATE_ARRAY 用于释放数组占用的内存。

第 19 行的宏 DEALLOCATE 用于释放内存。

第 23 行定义的结构体 String 用于存储字符串，这是为后面定义的 String 类型缓冲区做准备，马上会介绍。

第 28 行定义的结构体 CharValue 用于存储字符串对象中的字符串，字符串对象 ObjString 也是在后面介绍。因为是存储不定长度的字符串，因此在 CharValue 中 char start[0]数组长度为 0，就是用于柔性数组成员。数组长度由 length 控制，由于字符串的结束字符\0 不属于字符串的长度之内，故此值是除结束\0 之外的字符个数，这个得印象深刻，以后很多地方都涉及此数组中数据

仅供非商业用途或交流学习使用

的长度。

第 34 行的宏 DECLARE_BUFFER_TYPE 用于定义整个语言的核心存储结构：type##Buffer 结构体，它专用于存储类型为 type 的数据缓冲区，相当于简陋的 STL（估计是高攀了，脸红），在本系统中广泛用于存储符号、数值等。其中##是预处理器提供的字符串连接符，即若 type 为 String，生成的结构体名字便是 StringBuffer。结构体中的 datas 是 type*数组，即元素为指针类型 type*的数组，此数组用于存储数据。count 是指数组 datas 中已使用的元素个数。capacity 是数组 datas 中元素总个数，即数组的容量——包括已使用的和未使用的元素个数，此数组会由后面定义的一些方法在需要时扩容。

第 43～47 行定义的是一些操作 type##Buffer 结构体的方法的声明，它们的实现是由第 50 行定义的宏 DEFINE_BUFFER_METHOD 完成的。

第 51 行定义的方法 type##BufferInit 用于初始化缓冲区 buf，很简单。

第 56 行定义的方法 type##BufferFillWrite 用于往缓冲区 buf 中写入 fillCount 个类型为 type 的数据 data，第 58～59 行判断 buf 是否需要扩容，扩容是先调用 ceilToPowerOf2 计算新的容量，然后在第 64 行调用 memManager 扩容。第 66～70 行把 buf->count 当做数组索引往 datas 中写入数据，同时完成 count 自加 1 以更新下一次写入数据的索引地址。

第 73 行定义的方法 type##BufferAdd 是 type##BufferFillWrite 的封装，用于往 buf 中写入 1 个 data。

第 77 行定义的方法 type##BufferClear 用于清空 buf，分两步完成，先在第 79 行释放 buf->datas 的内存，然后在第 80 行初始化 buf。

第 83～90 行是声明 StringBuffer、IntBuffer、CharBuffer 和 ByteBuffer 及其配套的操作方法。习惯上复合数据类型首字符都用大写，因此定义了 Byte、Char 和 Int。

第 92 行定义的 ErrorType 是错误类型，用于不同类型的报错。再后面是相关报错的宏定义，都是调用函数 errorReport 实现的。

2.4 定义虚拟机结构（本节源码 stepByStep/c2/b）

词法分析器和虚拟机也有一些关联，比如词法分析器中若有申请内存的操作时便需要调用之前介绍过的 memManager 函数，memManager 第一个参数就是虚拟机指针 vm，因此词法分析器需要记录当前的是哪个 vm。也许有读者会问，解释器中的虚拟机还可以有多个？是的，这是为了将来能扩展为多实例的虚拟机做的准备（希望有那么一天），所有需要操作虚拟机的函数都要用参数指定操作哪一个虚拟机。而且，在虚拟机运行时需要通过词法分析器获得当前的编译单元，当然这是后话。所以，在此先定义好虚拟机的数据结构 struct vm，本节新建目录 vm，这正是虚拟机及相关核心代码的目录，请见 vm 目录下的 vm.h 文件。

stepByStep/c2/b/vm/vm.h

```
 1 #ifndef _VM_VM_H
 2 #define _VM_VM_H
 3 #include "common.h"
 4
 5 struct vm {
 6     uint32_t allocatedBytes;  //累计已分配的内存量
 7     Parser* curParser; //当前词法分析器
 8 };
 9
10 void initVM(VM* vm);
```

```
11 VM* newVM(void);
12 #endif
```

此时第 5 行定义的虚拟机结构 struct vm 只是临时为满足词法分析器运行而准备的，它并不完整，以后再逐步使其丰满。成员 allocatedBytes 是由 memManager 中使用的，用来汇总分配的总内存量。

我们将来实现的词法分析器是用结构 struct parser 定义的，在之前介绍过的文件 common.h 中，有 typedef struct parser Parser 把它定义为 Parser，我们说过了，复合数据结构一律首字符大写。因此在 struct vm 第 7 行中用 Parser*声明了当前词法分析器的指针 curParser，注意这里指的是“当前”词法分析器，一个源码要用一个词法分析器来解析，因此编译到不同的文件时词法分析器也不同，后面我们就会知道，这是由于支持模块导入造成的，全局上只有一个虚拟机，有多个不同的词法分析器。同样在 common.h 中也有 typedef struct vm VM，因此以后都是用 VM 来表示虚拟机。是否会有读者在想:“不是首字符大写吗？怎么都是大写？”。是这样的，VM 是 Virtual Machine 的首字母缩写，如果按照小驼峰式命名就是 vM，因此“小驼峰式命名+复合结构首字符大写”就成了 VM。

下面看下 vm.c。

stepByStep/c2/b/vm/vm.c

```
 1 #include <vm.h>
 2 #include <stdlib.h>
 3 #include <utils.h>
 4
 5 //初始化虚拟机
 6 void initVM(VM* vm) {
 7     vm->allocatedBytes = 0;
 8     vm->curParser = NULL;
 9 }
10
11 VM* newVM() {
12     VM* vm = (VM*)malloc(sizeof(VM));
13     if (vm == NULL) {
14         MEM_ERROR("allocate VM failed!");
15     }
16     initVM(vm);
17     return vm;
18 }
```

vm.c 中就两个函数，由于虚拟机 VM 定义尚不完整，因此这两个函数也都是不完整的。newVM 用于创建新的虚拟机，然后调用 initVM 初始化虚拟机。

2.5 实现源码读取（本节源码 stepByStep/c2/c）

词法分析器首先要做的就是读取源码文件以获得源码字符串，这个工作还是很简单的。一些核心的部件会放在 vm 目录中，读取文件的函数定义在 vm 目录中的文件 core..c 中。

stepByStep/c2/c/vm/core.c

```
1 #include "core.h"
2 #include <string.h>
```

```
 3 #include <sys/stat.h>
 4 #include "utils.h"
 5 #include "vm.h"
 6
 7 char* rootDir = NULL;    //根目录
 8
 9 //读取源代码文件
10 char* readFile(const char* path) {
11     FILE* file = fopen(path, "r");
12     if (file == NULL) {
13         IO_ERROR("Could't open file \"%s\".", path);
14     }
15
16     struct stat fileStat;
17     stat(path, &fileStat);
18     size_t fileSize = fileStat.st_size;
19     char* fileContent = (char*)malloc(fileSize + 1);
20     if (fileContent == NULL) {
21         MEM_ERROR("Could't allocate memory for reading file \"%s\".\n", path);
22     }
23
24     size_t numRead = fread(fileContent, sizeof(char), fileSize, file);
25     if (numRead < fileSize) {
26        IO_ERROR("Could't read file \"%s\".\n", path);
27     }
28     fileContent[fileSize] = '\0';
29
30     fclose(file);
31     return fileContent;
32 }
```

第 7 行的 rootDir 初始化为 NULL，它表示源码所在的根目录，其值是在另外一个文件 cli.c 中设置的，解释器运行时会获得源码文件的路径并把路径写入 rootDir，将来完成 cli.c 时你就明白了，这里先提前准备好。

函数 readFile 是用于读取源码，参数 path 是源码的路径。先在第 11 行调用 fopen 函数获得源码的句柄为后面的读取做准备，在第 17 行调用 stat 获得源码文件的大小后，在第 19 行通过 malloc 为源码字符串分配内存，多申请的 1 字节是为了字符串结束符\0。最后在第 24 行调用 fread 获得源码串到 fileContent，第 28 行人为加上结尾的\0。此时源文件便被读取到 fileContent 中成为一个大大的字符串。补充一下，C 编译器会将字符转换为 ascii 码，\0 的 ASCII 码是 0，因此直接写成 0 也行，只是编译器不需要转换了。但还是建议显式用\0 表示，这样语义较明确，我的老师说编程中还是少炫奇技淫巧，否则会被其他阅读源码的人鄙视。

一般情况下，除了包含了新的头文件和头文件中的函数声明外，其他代码若有变更，我都会贴出差异的部分，让大家在书上都能找到。这是为了照顾一些电脑环境不方便的读者，有一个课余学习计算机的大一学生跟我说过，他希望源码在书上都能找到，因为他妈妈怕他不务正业，所以不给他买电脑。

下面是头文件 core.h，只是一些声明，不说了。

stepByStep/c2/c/vm/core.h

```
1 #ifndef _VM_CORE_H
```




```
2 #define _VM_CORE_H
3 extern char* rootDir;
4 char* readFile(const char* sourceFile);
5 #endif
```

2.6 unicode 与 UTF-8

本节介绍一些字符集编码的内容，终于不用写码了。

2.6.1 什么是 unicode

unicode 是一种编码，8 位的字节一共可以组合出 256（2 的 8 次方）种不同的状态，人们把编号从 0 开始的 32 种状态（即编码为 0～31）与终端、打印机等设备“约定”为一些“控制”用途，一旦终端、打印机等设备接收到了这些“约定好的”的字节时，就要做一些“约定好的”的动作，比如遇上 0xA，终端就换行，这样工作很好，于是就把这些 0x20 以下的字节状态称为“控制码”。又把所有的空格、标点符号、数字、大小写字母分别用连续的字节状态表示，一直编号到了第 127 号，这样计算机就可以用不同字节来存储英语的文字了。这个方案叫作 ASCII。

后来 ISO（国际标准化组织）重新制订了一套包括地球上所有文化、所有字母和符号的编码，这就是“Universal Multiple-Octet Coded Character Set”，俗称 unicode，简称 UCS，即通用字符集。

unicode 开始制订时，计算机的存储器容量极大地发展了，空间再也不成为问题，于是 ISO 就规定必须用两个字节，也就是 16 位二进制来统一“表示”所有的字符，这 16 位二进制的数值就被称为“code point”，即码点，也就是说，码点是 unicode 给每个字符的编号。就像 ASCII 码表中的字符都有个编号一样，0x41 就相当于大写字母“A”的码点，只不过码点是 unicode 中的概念。对比一下，最初的 ASCII 码是 7 位，经过扩展后为 8 位，因此 ASCII 编号范围是 0x0～0xff，即两位十六进制数字表示一个 ASCII 字符。unicode 是 16 位，因此范围是 0x0～0xffff，即 4 位十六进制表示一个 unicode 字符，比如“汉”的 unicode 码点是 0x6C49。注意，这里是用两字符来“表示”一个字符，即用两字节来为一个字符编号，并不是也用两字节去“存储”它，你可以用大于两字节的空间去存储这两字节的数据，就像 32 位下用 4 字节存储整数 1 一样，尽管 1 位就够了（下节会提到这个问题）。

对于 ASCII 里的那些“半角”字符，unicode 保持其原编码不变，只是将其长度由原来的 8 位扩展为 16 位。由于“半角”英文符号只需要用到低 8 位，所以其高 8 位永远是 0，因此这种方案在保存英文文本时会多浪费一倍的空间。这时候早期的程序员开始发现一个奇怪的现象：他们的 strlen 函数靠不住了，一个汉字不再是相当于两个字符了，而是一个。其实这是肯定的，从 unicode 开始，无论是半角的英文字母，还是全角的汉字，它们都是统一的“一个字符”，同时也都是统一的“两个字节”，请注意“字符”和“字节”两个术语的不同，“字节”是一个 8 位的物理存储单元，而“字符”则是一个相关的符号。在 unicode 中，一个字符就是两个字节。一个汉字算两个英文字符的时代快过去了。

2.6.2 什么是 UTF-8

unicode 虽然能容纳上百万数量的字符，但是它只是一个巨大的字符集，仅仅规定了每个符号的二进制代码“表示”，却没有制定细化的“存储”规则，因此它仅限于“概念”，没有具体落实该怎么“实现”，因此到目前为止还只是纸上谈兵。这导致 unicode 有不少问题，比如当用 3 个字

节存储一个 unicode 字符时，它同时也可以被理解为存储了 3 个 1 字节大小的 ASCII 码，这有二义性。另外我们之前知道 ASCII 码只需要一个字节，但是，如果 unicode 规定每个字符都用 3 个字节来存储的话，那岂不是额外浪费两个字节的空间？所有这些未细化的问题都将导致 unicode 的不一致性，因此 unicode 在很长一段时间内无法推广。

直到互联网的普及，强烈要求出现一种统一的编码方式，为解决 unicode 如何在网络上传输的问题，也就是到了 unicode 该如何具体实现的地步了，UTF（UCS Transfer Format）标准出现了，其中 UCS 是指上节提到的 unicode 的简称，即通用字符集。其中较著名的两种 UTF 标准分别是 UTF-8 和 UTF-16。顾名思义，UTF-8 就是每次传输 8 位数据，而 UTF-16 就是每次 16 位。UTF-8 就是在互联网上使用最广的一种 unicode 的“实现”方式，这是为传输而设计的编码。很多读者之前都搞不清楚 unicode 和 UTF-8 的区别。unicode 是编码方案，既然称方案就表示不是具体实施，属于“行动”的指导方针，而 UTF-8 是 unicode 的一种实现，是 unicode 具体的“行动”。其他实现方式还包括 UTF-16 和 UTF-32，不过在互联网上基本不用。重复一遍，这里的关系是，UTF-8 是 unicode 的实现方式之一。

2.6.3　UTF-8 编码规则

UTF-8 最大的一个特点就是它是一种变长的编码方式。它可以使用 1～4 个字节表示一个符号，根据不同的符号而变化字节长度。UTF-8 的编码规则很简单，只有两条。

（1）对于单字节的符号，字节的第一位设为 0，后面 7 位为这个符号的 unicode 码。因此对于英语字母，UTF-8 编码和 ASCII 码是相同的。

（2）对于 n 字节的符号（n>1），第一个字节的前 n 位都设为 1，第 n+1 位设为 0，后面字节的前两位一律设为 10。剩下的没有提及的二进制位，全部为这个符号的 unicode 码。

表 2-1 是 UTF-8 编码规则总览，字母 x 表示可用编码的位。

表 2-1

Unicode 符号范围（十六进制）	UTF-8 编码方式（二进制）
0000 0000-0000 007F	0xxxxxxx
0000 0080-0000 07FF	110xxxxx 10xxxxxx
0000 0800-0000 FFFF	1110xxxx 10xxxxxx 10xxxxxx
0001 0000-0010 FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

对照表 2-1 细说下 UTF-8 编码规则。

UTF-8 是变长编码方式，即存储长度并不固定，长度是 1～4 字节，因此得有个方法标记本 UTF-8 编码占据了多少个字节。第一个字节称为高字节，其他字节称为低字节。本编码用“几个字节”表示，高字节中“前几位”就都是 1，然后接个 0，低字节的高 2 位是固定的 10。为叙述方便，把高字节和低字节中那些用以标识 UTF-8 特征的位（不能存储数据的位）暂称为标记位。假设一个 UTF-8 编码占用 n 个字节，高字节的高 n 位就都是 1，第 n+1 位是 0。低字节的高位均以 10 开头。除了各字节高位的标记位之外的其他位才是真正存储数据的位，暂称数据位。

为什么第 n+1 位是 0？UTF-8 编码无论用多少字节，都必须要告之有几个字节，显然，高字节的标记位是随字节长度的不同而变化的（低字节的标记位始终是 10，长度不变）。比如按照规则，2 字节的 UTF-8 编码为了表示一共占了 2 字节，必须在高字节的高位用两个 1 表示占据了 2 个字节，但是如何正确读到这两个 1 呢？毕竟该字节的其余位（即数据位）也可以存储数据，即其余位也可以是 1。为了区分数据位的 1 和标记位的 1，在它们之间用 0 分隔。因此解码 UTF-8 时，只要从高字节的最高位开始往低位读，读入连续的 1，直到遇到 0 为止，有几个 1 就表示本

UTF-8 编码所表示的数据占据了多少字节。比如对于 2 字节的 UTF-8 编码来说，其高字节的高 3 位必须是 110，高字节的其余 5 位才用于存储数据，低字节的高 2 位是固定的 10。同理，3 字节的 UTF-8 编码为了表示占据的是 3 个字节，高字节的高 4 位是 1110，低字节的高 2 位始终是固定的 10。

为什么 2 字节以上（包括 2 字节）的 UTF-8 编码，低字节的高 2 位始终是固定的 10 呢？这是因为 ASCII 码应用最广泛，任何编码为了发展都会主动兼容它，所以 UTF-8 也同样兼容 ASCII 码，但是 UTF-8 的编码规则和 ASCII 不同，必须要特殊处理 ASCII 码。任何编码在底层上都是二进制字节流，解码器在获得 1 字节的二进制数据时，如何知道这是 ASCII 码呢？还是 UTF-8 编码的高字节或低字节？所以 UTF-8 首先要做的是在二进制字节上必须与 ASCII 区分开来，即在这 1 字节数据上做标记，通过标记就知道这是 UTF-8 编码还是 ASCII 码。ASCII 是单字节，2 字节以上的数据用 UTF-8 编码才有意义，否则 1 个字节就够用的话就直接用 ASCII 了。ASCII 码范围是 0～127，因此其最高位是 0，而 2 字节以上的 UTF-8 编码其高字节最高位是 1，这样高字节已经可以和 ASCII 区分了，那么低字节如何和 ASCII 区分呢？有读者说了，只要 UTF-8 编码中低字节最高位也不为 0 就可以了，即只要是 1 就行。那我反问你，既然用 1 就行了，那 UTF-8 为什么要用 10 呢？我的答案是，因为如果只要求最高位是 1，那么就有可能和高字节混淆，比如二进制 11101101，这是 UTF-8 的高字节还是低字节？无法区分。那么，最不可能成为 UTF-8 高字节标记位的就是 10，如果 10 是高字节的标记位，按照 UTF-8 编码规则，说明 UTF-8 编码只用了 1 字节，显然这是矛盾的，因为在 UTF-8 中 1 字节的字符用其兼容的 ASCII 码表示，因此才说 UTF-8 至少要 2 字节以上才有意义，1 字节纯粹是为了兼容 ASCII 码，所以采用 10 作为低字节的标识位才是最合适的。既然标识位 10 只能出现在低字节的高 2 位，那么反过来说，UTF-8 编码中以 10 开头的都是低字节。低字节有了这个特性便具备了校验的能力，比如读取 UTF-8 编码的高字节后，确保后面的低字节必须以 10 开头才是正确的 UTF-8 编码。

由于 UTF-8 编码是变长的，因此不同字节的 UTF-8 编码表示的数值范围也不同，故可以通过判断数值的范围来反推该数字用 UTF-8 编码需要多少个字节。比如 2 字节 UTF-8 编码的标记位共占了 5 位，分别是高字节的高 3 位 110 和低字节的高 2 位 10，其他 11 位用于存储数据，这 11 位的最大值便是 0x7ff，因此数值小于或等于 0x7ff 的话，按照 UTF-8 就编码为 2 字节。同理 3 字节 UTF-8 编码的标记位共占了 8 位，分别是高字节的高 4 位 1110，中间字节的高 2 位 10 和最低字节的高 2 位 10，其余 16 位用于存储数据，故数值最大是 0xffff，因此小于 0xffff 的数值按照 UTF-8 编码只需要 3 字节。小于等于 127 的数值用单字节的 ASCII 码即可，因为其高位为 0，这与 UTF-8 编码是可区分的，前面说过了这是为了兼容 ASCII。UTF-8 解码器在解码时会注意到：最高位不是 1 的字节便是 ASCII 编码。

2.6.4 实现 UTF-8 编码、解码（本节源码 stepByStep/c2/d）

了解了 UTF-8 编码规则后就要进行编码和解码了。本节在 include 目录下创建文件 unicodeUtf8.c。

stepByStep/c2/d/include/unicodeUtf8.c

```
1 #include "unicodeUtf8.h"
2 #include "common.h"
3
4 //返回 value 按照 UTF-8 编码后的字节数
5 uint32_t getByteNumOfEncodeUtf8(int value) {
6     ASSERT(value > 0, "Can't encode negative value!");
7
8     // 单个 ASCII 字符需要 1 字节
```

```
    if (value <= 0x7f) {
        return 1;
    }

    //此范围内数值编码为 UTF-8 需要 2 字节
    if (value <= 0x7ff) {
        return 2;
    }

    //此范围内数值编码为 UTF-8 需要 3 字节
    if (value <= 0xffff) {
        return 3;
    }

    //此范围内数值编码为 UTF-8 需要 4 字节
    if (value <= 0x10ffff) {
        return 4;
    }

    return 0;  //超过范围返回 0
}

//把 value 编码为 UTF-8 后写入缓冲区 buf，返回写入的字节数
uint8_t encodeUtf8(uint8_t* buf, int value) {
    ASSERT(value > 0, "Can't encode negative value!");

//按照大端字节序写入缓冲区
    if (value <= 0x7f) {    // 单个 ASCII 字符需要 1 字节
        *buf = value & 0x7f;
        return 1;
    } else if (value <= 0x7ff) {  //此范围内数值编码为 UTF-8 需要 2 字节
        //先写入高字节
        *buf++ = 0xc0 | ((value & 0x7c0) >> 6);
        // 再写入低字节
        *buf = 0x80 | (value & 0x3f);
        return 2;
    } else if (value <= 0xffff) { //此范围内数值编码为 UTF-8 需要 3 字节
        // 先写入高字节
        *buf++ = 0xe0 | ((value & 0xf000) >> 12);
        //再写入中间字节
        *buf++ = 0x80 | ((value & 0xfc0) >> 6);
         //最后写入低字节
        *buf = 0x80 | (value & 0x3f);
        return 3;
    } else if (value <= 0x10ffff) { //此范围内数值编码为 UTF-8 需要 4 字节
        *buf++ = 0xf0 | ((value & 0x1c0000) >> 18);
        *buf++ = 0x80 | ((value & 0x3f000) >> 12);
        *buf++ = 0x80 | ((value & 0xfc0) >> 6);
        *buf = 0x80 | (value & 0x3f);
        return 4;
```

```
    }

    NOT_REACHED();
    return 0;
}

//返回解码 UTF-8 的字节数
uint32_t getByteNumOfDecodeUtf8(uint8_t byte) {
    //byte 应该是 UTF-8 的最高 1 字节，如果指向了 UTF-8 编码后面的低字节部分则返回 0
    if ((byte & 0xc0) == 0x80) {
        return 0;
    }

    if ((byte & 0xf8) == 0xf0) {
        return 4;
    }

    if ((byte & 0xf0) == 0xe0) {
        return 3;
    }

    if ((byte & 0xe0) == 0xc0) {
        return 2;
    }

    return 1;   //ascii 码
}

//解码以 bytePtr 为起始地址的 UTF-8 序列，其最大长度为 length，若不是 UTF-8 序列就返回-1
int decodeUtf8(const uint8_t* bytePtr, uint32_t length) {
    //若是 1 字节的 ascii:  0xxxxxxx
    if (*bytePtr <= 0x7f) {
        return *bytePtr;
    }

    int value;
    uint32_t remainingBytes;

    //先读取高 1 字节
    //根据高字节的高 n 位判断相应字节数的 UTF-8 编码
    if ((*bytePtr & 0xe0) == 0xc0) {
        //若是 2 字节的 utf8
        value = *bytePtr & 0x1f;
        remainingBytes = 1;
    } else if ((*bytePtr & 0xf0) == 0xe0) {
        //若是 3 字节的 utf8
        value = *bytePtr & 0x0f;
        remainingBytes = 2;
    } else if ((*bytePtr & 0xf8) == 0xf0) {
        //若是 4 字节的 utf8
```

```
109             value = *bytePtr & 0x07;
110             remainingBytes = 3;
111         } else {    //非法编码
112             return -1;
113         }
114
115         //如果 UTF-8 被斩断了就不再读过去了
116         if (remainingBytes > length - 1) {
117             return -1;
118         }
119
120         //再读取低字节中的数据
121         while (remainingBytes > 0) {
122            bytePtr++;
123            remainingBytes--;
124            //高 2 位必须是 10
125            if ((*bytePtr & 0xc0) != 0x80) {
126                return -1;
127            }
128
129            //从次高字节往低字节，不断累加各字节的低 6 位
130            value = value << 6 | (*bytePtr & 0x3f);
131         }
132         return value;
133 }
```

第 5 行的函数 getByteNumOfEncodeUtf8 用于返回数值 value 按照 UTF-8 编码后的字节数，函数的原理很简单。根据数值 value 的大小反推 UTF-8 编码需要的字节数。经过 UTF-8 编码的字节中的位可分为标记位和数据位，经过之前的分析读者都已知道标记位仅是编码规则上的标识需要，真正用于存储数据的部分是数据位，UTF-8 编码长度不同则存储数据的大小不同，因此根据待编码的数据大小便能判断编码长度，也就是编码后需要占用的字节数。例如，UTF-8 编码若为 3 字节，除了高字节高 4 位的 1110、中字节高 2 位的 10 和最低字节高 2 位的 10 一共 8 位的标识位外，还剩下 16 位即 2 字节的数据位，2 的 16 次方是 65536，即表示的数字范围是 0～65535。总结如表 2-2 所列。

表 2-2

待编码的数值		UTF-8 编码字节长度
十六进制	十进制	
<= 0x7f	<= 127	1
<= 0x7ff	<= 2047	2
<= 0xffff	<= 65535	3
<= 0x10ffff	<= 1114111	4

函数就是按照表 2-2 实现的。

第 32 行定义的函数 encodeUtf8 用于把 value 编码为 UTF-8 后，写入缓冲区 buf，返回写入的字节数。编码就是按照 UTF-8 规则把数据装入 UTF-8 的高字节和低字节中。例如，长度为 2 的 UTF-8 编码形式为“110xxxxx 10xxxxxx”，假如把数据编码为 2 字节的 UTF-8 编码，只要把数据装入其中的 x 所在的位即可，这就是编码的原理。

本函数是按照大端字节序来编码的，即先写数据的高字节，再写低字节。编码的思路还是根据 value 的大小来判断。

第 36 行针对单字节的 ASCII 码，若是单字节，存储方式与 ASCII 一样，用 7 位二进制 1 做“位与”运算以获得 1 字节的 ASCII 码。编码写入 buf。

后面的部分涉及 UTF-8 编码的高字节和低字节，高字节要写入字节数标识位，低字节要写入标识符 10。如第 39 行代码为小于等于 0x7ff 的 value 编码，0x7ff 的二进制，高字节是 00000111，低字节是 11111111，这是 2 字节 UTF-8 编码的容量上限，因此高字节中左边 5 个 0 所在的位在实际 UTF-8 编码中是不存在的，2 字节只能容下低 11 位，即那些用 1 表示的位。现在要把这低 11 位装入 2 字节的 UTF-8 编码中的数据位，即装入高字节 110xxxxx 和低字节 10xxxxxx 中 x 所在的位。注意，此处高字节只有 5 位可用，也就是说要把 value 的“实际”高 5 位装入，即 value 低 11 位中的高 5 位，若以低字节第 0 位开始算起，value 的实际高 5 位是第 10～6 位，即“11111，111111”中逗号左边的高 5 位，要获得左边的高 5 位，只要把这 11 位右移 6 位即可。下面先在第 41 行写入编码后的高字节，用 0x7c0（二进制形式是 11111000000）与 value 做“与运算”过滤出左边的 5 位，然后右移 6 位，即获得了左边 5 位。接下来构建 UTF-8 高字节标识位，由于是 2 字节的 UTF-8 编码，前 2 位应该是 110，扩展到字节应该是 0xc0，因此最后与 0xc0 做“或运算”合并为 UTF-8 高字节，写入 buf 后，使 buf 自加 1，更新为下一个可写地址。下面写入 UTF-8 编码的低字节，低字节除了标识位 10 外只有 6 位可用，即把原值“11111，111111”中逗号右边的低 6 位写入即可。UTF-8 低字节标识位是 10，扩展为字节 0x80，在第 43 行用 0x3f 与 value 做“与运算”过滤出低 6 位（即上行逗号右边的低 6 位）后再与 0x80 做“或运算”合并为 UTF-8 的低字节，写入 buf 后，返回 2 字节。

第 45 行和第 53 行分别处理 3 字节和 4 字节的 UTF-8 编码，编码形式分别为“1110xxxx 10xxxxxx 10xxxxxx”和“11110xxx 10xxxxxx 10xxxxxx 10xxxxxx”，原理同 2 字节的编码相同，不再赘述。

第 66 行的函数 getByteNumOfDecodeUtf8 返回解码 UTF-8 编码所需要的字节数，参数 byte 是 UTF-8 编码的高字节。函数的原理是判断 UTF-8 高字节的标识位中 1 的个数，有几个 1 就表示有几字节，这可以用一字节与之做“与运算”试探。

第 68 行用 0xc0 与 byte 做“与运算”来判断，如果结果为 0x80，说明这是“与”到了低字节中的标识位 10，对于 UTF-8 编码后面的低字节则返回 0。

第 72 行用 0xf8 与 byte 做“与运算”来判断，如果结果为 0xf0，说明编码 UTF-8 高字节的标识位是 11110，因此返回 4 字节。下同，不再赘述。

第 88 行的函数 decodeUtf8 解码一个经过 UTF-8 编码的字符序列，参数 bytePtr 指向该字符序列，即指向 UTF-8 编码的高字节或 ASCII 码，序列最大长度为 length 若不是 UTF-8 序列就返回−1。解码就是编码的逆过程，判断是多少字节的编码，然后把数据部分读出来，以 2 字节为例，解码就是把“110xxxxx 10xxxxxx”中的 x 读出来。

第 90 行判断若小于等于 0x7f 即 127，说明是单字节的 ASCII 码，直接返回。

下面就是处理多字节的情况了，先读取高字节，根据高字节的高 n 位判断相应字节数的 UTF-8 编码。

若是 2 字节的 UTF-8 编码，即“110xxxxx 10xxxxxx”，那么高 3 位必须为 110，因此第 96 行用 0xe0 即二进制 11100000 与高字节*bytePtr 做“与运算”，结果若为 0xc0 即二进制 11000000，也就是高 3 位为 110，说明此 UTF-8 编码正是 2 字节。高字节的数据部分是低 5 位，因此在第 101 行获取高字节的低 5 位存入 value。变量 remainingBytes 用于记录剩下未读的字节数，即低字节的个数，因此对于 2 字节的 UTF-8 编码，由于已经读过了高字节，因此使 remainingBytes 为 1。下

面至 113 行原理相同。

参数 length 是帮助判断 UTF-8 编码是否完整，有可能数据流不完整被斩断，UTF-8 编码长度是“最高 1 字节+剩余 n 字节（1≤n≤3）”，剩余的 n 字节是要从后面的数据流中读取，其中 length 就表示包括最高字节在内的数据流的字节数，因此数据若完整的话，剩余 n 字节要小于等于 length-1，其中-1 是去掉最高 1 字节即数据流中除最高 1 字节外剩下的字节数。因此在第 116 行判断 remainingBytes 若大于 length-1，说明此 UTF-8 序列被斩断了。

第 121 行开始读取剩余的低字节，低字节的高 2 位标识符必须是 10，因此在第 125 行把*bytePtr 和 0xc0 做“与运算”来判断这个特性，若结果为 0x80 才正确，否则报错。

第 130 行开始不断累加各低字节的数据位即低 6 位。

下面是 unicodeUtf8.h 文件。

stepByStep/c2/d/include/unicodeUtf8.h

```
1 #ifndef _INCLUDE_UTF8_H
2 #define _INCLUDE_UTF8_H
3 #include <stdint.h>
4 uint32_t getByteNumOfEncodeUtf8(int value);
5 uint32_t getByteNumOfDecodeUtf8(uint8_t byte);
6 uint8_t encodeUtf8(uint8_t* buf, int value);
7 int decodeUtf8(const uint8_t* bytePtr, uint32_t length);
8 #endif
```

2.7 实现词法分析器 parser（本节源码 stepByStep/c2/e）

2.7.1 lex 和 token

这两个都是用在词法分析中的术语，容易搞混，我在第 0 章中已经解释了它们的意义，在此简略回顾下。

lex 是 lexeme 的缩写，是词汇、单词的意思，也就是字符串本身的意义，即字符串字面量。比如“abc”就是 lex，它是词法分析器的输入，即需要词法分析器直接识别的原始材料，主要是供记法分析器使用。

token 是符号、象征的意思，因此它肯定不是最直接的字面量。它“象征”一个单词（lex），是单词的另一个符号化表示。token 是词法分析器的输出，即读取 lex 后的产出物，产出的东西很多，因此这个产出物通常是一个复合结构，里面包含了 lex 及 lex 的长度，还包括 lex 所在的行号等。token 是供语法分析器使用的，用以校验语法，即是否符合产生式。

2.7.2 字符串和字符串内嵌表达式

字符串读者都比较熟悉，它是用双引号括起来的一串字符，比如字符串“hello,world!”。下面先介绍字符串内嵌表达式。字符串内嵌表达式就是在字符串中嵌入的代码，该代码的执行结果会转换为字符串与其周边的字符串相连接。以下简称为内嵌表达式。

我们支持的内嵌表达式格式为，“[可有可无的字符串]%（代码）[可有可无的字符串]”，代码必须放在%()中。比如源码串“ab%(1+2)c”，此字符串就相当于 ab3c。也许有读者一眼就看出这是在运行时合成的结果，因此为什么要在词法分析阶段说这个呢？尽管这是在运行时发生的事，但还是需要在词法分析时先识别到这种情况，才能在语法分析阶段做语法检查的同时生成相应的指令。强调一下，这是在字符串中嵌入代码，因此%()也必须存在于双引号“”中，在双引号外的

%()就被认为非法。内嵌表达式只支持一级的嵌入，如“ab%(1+2)c%(3)”是允许的，嵌套多层的就不允许了，如 2 层嵌套“ab%(1+“%(2+3)”)”就不支持了，词法分析器会报错。

由于有了内嵌表达式，为了语法分析（不是词法分析）时处理方便，字符串类型变成了两种：一种是前面说过的用双引号“”括起的普通字符串，另一种是内嵌表达式所属字符串的末尾部分，比如“a %(b) c %(d) e”中的 e 就被视为字符串，注意有空格，空格也是字符串的组成部分。这个例子中加空格的目的就是给读者加深印象，在词法分析中，单词的识别不是按照空格去拆分的，而是按照文法规则。尽管变量名中间不能有空格，似乎“显得是”空格把变量名拆分了，但那是变量名的文法规则所做的限制，变量名只能由下划线、英文字母和数字组成，但数字不能作为变量名的开头字符。因此，是变量名中不能包括空格，并不是空格作为分界符。当然，变量名中包括空格也不合适，似乎是把变量名分成了两半儿，变量本身就是代表某个完整的个体，与看似被分成两半的变量名矛盾。

语法分析器的工作是从词法分析器那里接收 token，根据不同的 token 做出相应的规则匹配并生成相应的目标指令。为了语法分析器方便处理内嵌表达式，需要由词法分析器生成内嵌表达式的 token，因此包含内嵌表达式的字符串被词法分析器拆分成两类 token：内嵌表达式的 token 和普通字符串 token。这两类 token 在我们的系统中分别被定义为 TOKEN_INTERPOLATION 和 TOKEN_STRING，token 定义在文件 parse.h 中，很快就会见到它。举个例子，对于“a %(b) c %(d) e”来说：

a 是 TOKEN_INTERPOLATION，注意不包括%（和），词法分析时会跳过这些。

b 是变量名的 token，在我们系统中被定义为 TOKEN_ID。

c 是 TOKEN_INTERPOLATION。

d 是 TOKEN_ID。

e 是 TOKEN_STRING。

读者注意到，TOKEN_INTERPOLATION 是%()左边的部分。也许有读者会问，内嵌表达式按照定义是由%()括起的部分，为什么词法分析器却认为它左面的部分才是内嵌表达式呢？其实这样拆分是非常合理的，语法分析器对字符串和代码的处理肯定是不一样的，由%()括起的是具有“能动性”的代码，它并不是字符串，因此需要被语法分析器当作代码来处理。在%()左边的部分是字符串，语法分析器和词法分析器约定好了，由词法分析器返回%()前面的部分作为内嵌表达式的 token 给语法分析器，以提醒语法分析器后面的部分该是代码了，语法分析器马上就会明白，马上要处理字符串中包含的代码了。

2.7.3 单词识别流程

在此说明一下，单词是指文法中可推导出的所有终结符（文法和终结符等概念可查看第 0 章），比如关键字 var、变量名 xxx、数字 123、小括号“(”等，简单一句话：语法正确的词汇就是单词，词法分析器就是根据文法规则识别这些单词。

传统教材介绍词法分析器时往往连同状态机一起介绍，原因是传统的词法分析器是用状态机的原理实现的，状态机从开始态到终态的转移就是识别单词的过程，鉴于第 0 章中我们已经讨论过它的弊端了，因此这里不采用状态机。

其实我们的实现也很普通，目前大部分词法分析器也是这样做的，即不设置状态，按照单词的规则去匹配，采取一读到底的方式，直到遇到不符合规则的字符。大概流程如图 2-2 所示。

图虽然简单，但是涵盖了所有类型单词的识别方式。

词法分析器的逻辑是从源码串每次读入一个字符，如果是空白字符就跳过，因为有可能在源码的开头就是用户的缩进。接下来判断字符。

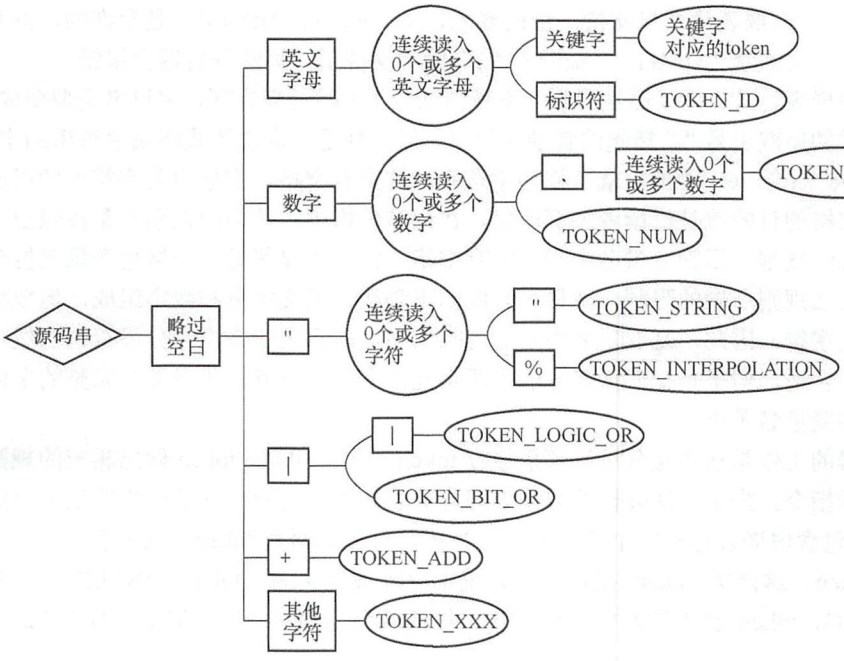


图 2-2

（1）如果是单字符单词，这是最简单的情况，比如+，由于我们并不支持++和+=，因此可直接判定 token 为 TOKEN_ADD（有关 TOKEN 的定义在下节的文件 parser.h 中）。

（2）如果字符是双字符单词的前缀，比如“位或运算符|”是“逻辑或运算符||”的前缀，那么在读入|后再向前查看一个字符，即再读入一个字符，如果下一字符不是|，那么就认定 token 为 TOKEN_BIT_OR，即代表识别的是|，如果下一字符是|，就代表识别的是||，那么 token 就为 TOKEN_LOGIC_OR。

（3）如果字符是“"”，说明是字符串的开始，由于我们支持内嵌表达式，下面分两种情况处理。

① 一种是普通的字符串，即在"后应该是一串字符，因此继续读取字符。直到遇到另一个“"”，识别结束，token 为 TOKEN_STRING。

② 另一种情况是内嵌表达式，内嵌表达式是用%()包围的代码，代码写在小括号中。如果读入若干字符（包括 0 个）后遇到%，说明是内嵌表达式，识别结束，token 为 TOKEN_INTERPOLATION。如果字符串中只有这 1 个内嵌表达式，那么“)”后面的部分被当成普通字符串，留待下一轮再识别。

（4）如果字符是数字，继续读取 0 个或多个数字，直到除小数点以外的非数字为止，整数识别结束，token 为 TOKEN_NUM。如果下一个字符是小数点，说明是浮点数，继续读取 0 个或多个数字，直到非数字为止，浮点数识别结束，token 同样为 TOKEN_NUM，也就是说数字类型不区分整数和浮点数。

（5）如果字符是英文字母，继续读入 0 个或多个英文字母、数字或下划线，直到遇到这 3 类字符之外的字符。此时读入的单词分为两类，一类是系统的关键字，即保留字，具体的 token 可查内部关键字表获得。另一类就是标识符，token 为 TOKEN_ID。

以上流程描述虽然简单，但已经覆盖了所有类型单词的识别策略，需要在实践中体会。

2.7.4　定义 token 和 parser

本节新建目录 parser，词法分析器头文件定义在 parser/parser.h，代码如下。

stepByStep/c2/e/parser/parser.h

```
#ifndef _PARSER_PARSER_H
#define _PARSER_PARSER_H
#include "common.h"
#include "vm.h"

typedef enum {
    TOKEN_UNKNOWN,
    // 数据类型
    TOKEN_NUM,              //数字
    TOKEN_STRING,           //字符串
    TOKEN_ID,               //变量名
    TOKEN_INTERPOLATION,    //内嵌表达式

    // 关键字(系统保留字)
    TOKEN_VAR,              //var
    TOKEN_FUN,              //fun
    TOKEN_IF,               //if
    TOKEN_ELSE,             //else
    TOKEN_TRUE,             //true
    TOKEN_FALSE,            //false
    TOKEN_WHILE,            //while
    TOKEN_FOR,              //for
    TOKEN_BREAK,            //break
    TOKEN_CONTINUE,         //continue
    TOKEN_RETURN,           //return
    TOKEN_NULL,             //null

    //以下是关于类和模块导入的 token
    TOKEN_CLASS,            //class
    TOKEN_THIS,             //this
    TOKEN_STATIC,           //static
    TOKEN_IS,               //is
    TOKEN_SUPER,            //super
    TOKEN_IMPORT,           //import

    //分隔符
    TOKEN_COMMA,            //,
    TOKEN_COLON,            //:
    TOKEN_LEFT_PAREN,       //(
    TOKEN_RIGHT_PAREN,      //)
    TOKEN_LEFT_BRACKET,     //[
    TOKEN_RIGHT_BRACKET,    //]
    TOKEN_LEFT_BRACE,       //{
    TOKEN_RIGHT_BRACE,      //}
    TOKEN_DOT,              //.
    TOKEN_DOT_DOT,          //..

    //简单双目运算符
    TOKEN_ADD,              //+
```

```
    TOKEN_SUB,              //-
    TOKEN_MUL,              //*
    TOKEN_DIV,              ///
    TOKEN_MOD,              //%

    //赋值运算符
    TOKEN_ASSIGN,           //=

    // 位运算符
    TOKEN_BIT_AND,          //&
    TOKEN_BIT_OR,           //|
    TOKEN_BIT_NOT,          //~
    TOKEN_BIT_SHIFT_RIGHT,  //>>
    TOKEN_BIT_SHIFT_LEFT,   //<<

    // 逻辑运算符
    TOKEN_LOGIC_AND,        //&&
    TOKEN_LOGIC_OR,         //||
    TOKEN_LOGIC_NOT,        //!

    //关系操作符
    TOKEN_EQUAL,            //==
    TOKEN_NOT_EQUAL,        //!=
    TOKEN_GREATE,           //>
    TOKEN_GREATE_EQUAL,     //>=
    TOKEN_LESS,             //<
    TOKEN_LESS_EQUAL,       //<=

    TOKEN_QUESTION,         //?

    //文件结束标记，仅词法分析时使用
    TOKEN_EOF               //EOF
} TokenType;

typedef struct {
    TokenType type;
    const char* start;
    uint32_t length;
    uint32_t lineNo;
} Token;

struct parser {
    const char* file;
    const char* sourceCode;
    const char* nextCharPtr;
    char curChar;
    Token curToken;
    Token preToken;

    //处于内嵌表达式之中时，期望的右括号数量
```

```
100     //用于跟踪小括号对儿的嵌套
101     int interpolationExpectRightParenNum;
102     VM* vm;
103 } ;
104
105 #define PEEK_TOKEN(parserPtr) parserPtr->curToken.type
106
107 char lookAheadChar(Parser* parser);
108 void getNextToken(Parser* parser);
109 bool matchToken(Parser* parser, TokenType expected);
110 void consumeCurToken(Parser* parser, TokenType expected, const char* errMsg);
111 void consumeNextToken(Parser* parser, TokenType expected, const char* errMsg);
112 uint32_t getByteNumOfEncodeUtf8(int value);
113 uint8_t encodeUtf8(uint8_t* buf, int value);
114 void initParser(VM* vm, Parser* parser, const char* file, const char* sourceCode);
115 #endif
```

第 6～82 行定义的 TokenType，每个单词都对应一类 token。

第 89 行定义的 Token 就是之前常说的词法分析器合成单词周边信息的复合结构 token。其中 start 指针指向源码串中单词的起始地址，length 是此单词的长度，lineNo 是单词所在源码中的行号。

第 91～103 行定义的 struct parser 是词法分析器的结构，在 common.h 中有代码 typedef struct parser Parser，同前面 VM 结构的理由一样，各种结构的定义相互包含依赖，这也是为了前向引用，因此以后是用 Parser 作为词法分析器。成员 file 通常指向源码文件名，便于指出当前正编译哪个文件，如果编译的源码不在文件中，比如是内部用源码定义的数据（也可以用源码来自举），此项赋予核心数据的名字。sourceCode 指向源码串，源码是存储在文件 file 中的字符串，将源码读出来后存储到某缓冲区，然后用 sourceCode 指向此缓冲区。nextCharPtr 指向 sourceCode 中下一个字符。curChar 是识别到的当前字符，curToken 是当前的 token，preToken 是前一个 token，在语法分析时，有时候我们需要前一个 token 的信息。interpolationExpectRightParenNum 这个名字最长的变量用于跟踪内嵌表达式%()中的小括号对儿的数量，用此变量来跟踪内嵌表达式中期望的右括号“)”的数量。有点像对儿匹配，出现一个“(”，那么肯定期望另一半“)”的出现。vm 指针表示本 parser 隶属于哪个 vm，在词法分析过程中若有内存分配需要指定 vm。

2.7.5 解析关键字及获取字符

parser.c 是词法分析器的实现，代码如下。

stepByStep/c2/e/parser/parser.c (1/6)

```
 1 #include "parser.h"
 2 #include <stdio.h>
 3 #include <stdlib.h>
 4 #include "common.h"
 5 #include "utils.h"
 6 #include "unicodeUtf8.h"
 7 #include <string.h>
 8 #include <ctype.h>
 9
10 struct keywordToken {
11     char*       keyword;
12     uint8_t     length;
```

```
    TokenType   token;
}; //关键字(保留字)结构

//关键字查找表
struct keywordToken keywordsToken[] = {
    {"var",       3,    TOKEN_VAR},
    {"fun",       3,    TOKEN_FUN},
    {"if",        2,    TOKEN_IF},
    {"else",      4,    TOKEN_ELSE},
    {"true",      4,    TOKEN_TRUE},
    {"false",     5,    TOKEN_FALSE},
    {"while",     5,    TOKEN_WHILE},
    {"for",       3,    TOKEN_FOR},
    {"break",     5,    TOKEN_BREAK},
    {"continue",  8,    TOKEN_CONTINUE},
    {"return",    6,    TOKEN_RETURN},
    {"null",      4,    TOKEN_NULL},
    {"class",     5,    TOKEN_CLASS},
    {"is",        2,    TOKEN_IS},
    {"static",    6,    TOKEN_STATIC},
    {"this",      4,    TOKEN_THIS},
    {"super",     5,    TOKEN_SUPER},
    {"import",    6,    TOKEN_IMPORT},
    {NULL,        0,    TOKEN_UNKNOWN}
};

// 判断 start 是否为关键字并返回相应的 token
static TokenType idOrkeyword(const char* start, uint32_t length) {
    uint32_t idx = 0;
    while (keywordsToken[idx].keyword != NULL) {
        if (keywordsToken[idx].length == length && \
                memcmp(keywordsToken[idx].keyword, start, length) == 0) {
            return keywordsToken[idx].token;
        }
        idx++;
    }
    return TOKEN_ID;
}

// 向前看一个字符
char lookAheadChar(Parser* parser) {
    return *parser->nextCharPtr;
}

//获取下一字符
static void getNextChar(Parser* parser) {
    parser->curChar = *parser->nextCharPtr++;
}

//查看下一个字符是否为期望的，如果是就读进来，返回 true，否则返回 false
```

```
63 static bool matchNextChar(Parser* parser, char expectedChar) {
64     if (lookAheadChar(parser) == expectedChar) {
65         getNextChar(parser);
66         return true;
67     }
68     return false;
69 }
70
71 // 跳过连续的空白字符
72 static void skipBlanks(Parser* parser) {
73     while (isspace(parser->curChar)) {
74         if (parser->curChar == '\n') {
75             parser->curToken.lineNo++;
76         }
77         getNextChar(parser);
78     }
79 }
```

第 10 行定义的 keywordsToken 是系统关键字结构，keyword 是关键字名字，length 是关键字长度，token 是 token 类型。

第 17 行定义的数组 keywordsToken 是系统的关键字表，所有关键字都存储到此数组中，数组元素类型就是上面定义的 keywordsToken。在词法分析流程图中识别的关键字就是在此数组中查到的。

第 40 行定义的函数 idOrkeyword 用于判断以 start 为起始，长度为 length 的单词是否为关键字，然后相应 token 类型。实现很简单，遍历 keywordsToken 数组一一比较，若不在关键字表（即数组 keywordsToken）中，说明是普通的标识符，则返回 TOKEN_ID。

第 53 行定义的函数 lookAheadChar 用于向前查看一个字符，就是返回指针 nextCharPtr 指向的字符。

第 58 行定义的函数 getNextChar 用于获取下一个字符，就是将 nextCharPtr 指向的字符放到当前字符 curChar 中，同时使 nextCharPtr 指向下一个字符。

第 63 行定义的函数 matchNextChar 用以匹配下一个期望的字符，如果下一个字符是参数 expectedChar 指定的字符，那么就读进该字符并返回 true，否则不读入任何字符并返回 false。

第 72 行定义的函数 skipBlanks 用于跳过空白字符，并在第 75 行同时累计行号。

2.7.6 解析标识符和 unicode 码点

stepByStep/c2/e/parser/parser.c (2/6)

```
81 //解析标识符
82 static void parseId(Parser* parser, TokenType type) {
83     while (isalnum(parser->curChar) || parser->curChar == '_') {
84         getNextChar(parser);
85     }
86
87     //nextCharPtr 会指向第 1 个不合法字符的下一个字符，因此-1
88     uint32_t length = (uint32_t)(parser->nextCharPtr - parser->curToken.start - 1);
89     if (type != TOKEN_UNKNOWN) {
90         parser->curToken.type = type;
91     } else {
```

```
92              parser->curToken.type = idOrkeyword(parser->curToken.start, length);
93      }
94      parser->curToken.length = length;
95  }
96
97  //解析 unicode 码点
98  static void parseUnicodeCodePoint(Parser* parser, ByteBuffer* buf) {
99      uint32_t idx = 0;
100     int value = 0;
101     uint8_t digit = 0;
102
103 //获取数值，u 后面跟着 4 位十六进制数字
104     while(idx++ < 4) {
105         getNextChar(parser);
106         if (parser->curChar == '\0') {
107             LEX_ERROR(parser, "unterminated unicode!");
108         }
109         if (parser->curChar >= '0' && parser->curChar <= '9') {
110             digit = parser->curChar - '0';
111         } else if (parser->curChar >= 'a' && parser->curChar <= 'f') {
112             digit = parser->curChar - 'a' + 10;
113         } else if (parser->curChar >= 'A' && parser->curChar <= 'F') {
114             digit = parser->curChar - 'A' + 10;
115         } else {
116             LEX_ERROR(parser, "invalid unicode!");
117         }
118         value = value * 16 | digit;
119     }
120
121     uint32_t byteNum = getByteNumOfEncodeUtf8(value);
122     ASSERT(byteNum != 0, "utf8 encode bytes should be between 1 and 4!");
123
124     //为代码通用，下面会直接写 buf->datas，在此先写入 byteNum 个 0，以保证事先有 byteNum 个空间
125     ByteBufferFillWrite(parser->vm, buf, 0, byteNum);
126
127     //把 value 编码为 UTF-8 后写入缓冲区 buf
128     encodeUtf8(buf->datas + buf->count - byteNum, value);
129 }
```

第 82 行定义的函数 parseId 用于解析标识符，变量名、函数名都称为标识符，标识符的规则是，由数字、字母和下划线组成，但首字符不能为数字。本函数是在 getNextToken 函数读入标识符的首字母后调用的，首字符不能为数字已经在 getNextToken 中判断过了，因此函数 parseId 是把标识符剩余的部分字符读进来。第 83 行判断如果是数字、字母或下划线，就循环调用 getNextChar 读字符。也许读者觉得奇怪，读进来的字符怎么没处理？似乎是“跳过”了，其实并没有丢掉它们，Token 结构中的成员 start 始终指向源码串中单词的首地址（稍后介绍 getNextToken 函数时）。length 是单词长度，确定一个单词只要知道 start 和 length 两项就够了。现在 start 已经有了，length 如何确定呢？while 循环中调用的 getNextChar 函数一直在更新 nextCharPtr，使其自加 1，直到遇到不属于数字、字母和下划线的字符后，循环结束。那时 nextCharPtr 是标识符的结尾，确切地说是指向标识符最后一个字符后面的字符。start 是标识符的起始，于是在第 88 行就可以利用这两项

来计算 length，然后在第 94 行把 length 存储到 curToken.length 中。参数 type 的值如果是 TOKEN_UNKNOWN，那么就在第 92 行调用 idOrkeyword 返回关键字或标识符的 token 类型以设置 curToken.type。如果 type 的值不为 TOKEN_UNKNOWN，即是已知的，那么就在第 90 行直接用它设置 curToken.type。由于本函数是在后面介绍的 getNextToken 中调用的，因此参数 type 的意义还需到那个环境中去了解，稍后会介绍。

就像可以用 ASCII 值 0x41 来表示大写字符 A 一样，字符还可以用 unicode 码点（code point）来表示，码点是 4 个十六进制数字，比如“字”的码点是 0x5B57，引用时为了与数字区分，这 4 位十六进制数字须以转义字符\u 做前缀，这样“字”的码点便是\u5B57。

第 98 行的函数 parseUnicodeCodePoint 用于解析 unicode 码点，将 unicode 码点按照 UTF-8 编码后写入 buf 指定缓冲区中。

第 104～119 行通过 while 循环调用 getNextChar 获取 length 个字符，根据数字字符的范围确认数字的值，第 118 行将各位的值转换为十进制后累计到 value 中。

第 121 行调用 getByteNumOfEncodeUtf8 获得 value 编码为 UTF-8 需要的字节数到 byteNum，紧接着在第 125 行调用 ByteBufferFillWrite 往 buf 中写入 byteNum 个 0 为存储第 128 行的编码准备好空间，因为第 128 行的 encodeUtf8 函数是直接往 datas 中写数据，并不是调用 ByteBufferAdd 方法，因此空间不足时无法为 buf 自动扩容，可查看前面介绍过的 encodeUtf8。

2.7.7 解析字符串、内嵌表达式、转义字符

本节主要是解析字符串，继续看代码。

stepByStep/c2/e/parser/parser.c (3/6)

```
130
131 //解析字符串
132 static void parseString(Parser* parser) {
133     ByteBuffer str;
134     ByteBufferInit(&str);
135     while (true) {
136         getNextChar(parser);
137
138         if (parser->curChar == '\0') {
139             LEX_ERROR(parser, "unterminated string!");
140         }
141
142         if (parser->curChar == '"') {
143             parser->curToken.type = TOKEN_STRING;
144             break;
145         }
146
147         if (parser->curChar == '%') {
148             if (!matchNextChar(parser, '(')) {
149                 LEX_ERROR(parser, "'%' should followed by '('!");
150             }
151             if (parser->interpolationExpectRightParenNum > 0) {
152                 COMPILE_ERROR(parser, "sorry, I don't support nest interpolate expression!");
153             }
154             parser->interpolationExpectRightParenNum = 1;
155             parser->curToken.type = TOKEN_INTERPOLATION;
```

```
156             break;
157         }
158
159         if (parser->curChar == '\\') {    //处理转义字符
160             getNextChar(parser);
161             switch (parser->curChar) {
162                 case '0':
163                     ByteBufferAdd(parser->vm, &str, '\0');
164                     break;
165                 case 'a':
166                     ByteBufferAdd(parser->vm, &str, '\a');
167                     break;
168                 case 'b':
169                     ByteBufferAdd(parser->vm, &str, '\b');
170                     break;
171                 case 'f':
172                     ByteBufferAdd(parser->vm, &str, '\f');
173                     break;
174                 case 'n':
175                     ByteBufferAdd(parser->vm, &str, '\n');
176                     break;
177                 case 'r':
178                     ByteBufferAdd(parser->vm, &str, '\r');
179                     break;
180                 case 't':
181                     ByteBufferAdd(parser->vm, &str, '\t');
182                     break;
183                 case 'u':
184                     parseUnicodeCodePoint(parser, &str);
185                     break;
186                 case '"':
187                     ByteBufferAdd(parser->vm, &str, '"');
188                     break;
189                 case '\\':
190                     ByteBufferAdd(parser->vm, &str, '\\');
191                     break;
192                 default:
193                     LEX_ERROR(parser, "unsupport escape \\%c", parser->curChar);
194                     break;
195             }
196         } else {    //普通字符
197             ByteBufferAdd(parser->vm, &str, parser->curChar);
198         }
199     }
200     ByteBufferClear(parser->vm, &str);
201 }
202
```

第 132 行定义的函数 parseString 用于解析字符串，第 133 行定义的 str 用于存储解析的字符，

第 134 行将其初始化。

字符串是用双引号括起的，如果在遇到右双引号之前源码串就结束了，这种情况就是字符串不完整。第 138 行处理不完整的字符串。

第 142 行处理字符串的结束，即遇到了“"”则认为字符串识别完成，将 token 设置为 TOKEN_STRING。

第 147 行处理内嵌表达式，在双引号引起的字符串若包含了字符%，则说明在字符串中包含了内嵌表达式。内嵌表达式的结构是%（代码），因此在%后的字符应该是“(”，因此第 148 行便去匹配下一个字符“(”，如果下一个不是“(”就在第 149 行报错。

当词法分析器第一次遇到%(时，注意，不包括右括号“)”，它会期待一个右括号，因此会将变量 interpolationExpectRightParenNum 置为 1，这正是第 154 行代码的作用。我们只支持一层内嵌表达式，即无法在内嵌表达式中再嵌套一个内嵌表达式。这里用变量 interpolationExpectRightParenNum 来跟踪这种情况，第 151 行判断这种情况，如果 interpolationExpectRightParenNum 大于 0，说明之前已经有了内嵌表达式了，并且正期待遇到%()中的“)”，如果第 147 行的当前字符是%，说明是嵌套了，因此在第 152 行报错。如果并未嵌套，在第 154 行将 interpolationExpectRightParenNum 设置为 1，并在第 155 行将 curToken.type 置为 TOKEN_INTERPOLATION。

源代码对词法分析器来说是文本字符串，因此我们在 sparrow 脚本源码中写入的转义字符也是两个字符，比如\n，在词法分析器眼里是\和 n。第 159～195 行处理各类转义字符，根据字符\后面的字符分别按 switch 的相应分支处理。由于我们是用 C 语言编写编译器，因此还是要按照 C 语言语法来写转义字符，对于 sparrow 脚本源码中的被视为“两个”字符组成的转义字符，我们在 C 代码中要将其翻译为“一个”字符的转义字符，比如在第 175 行将 sparrow 中连续的\和 n 翻译为\n，调用 ByteBufferAdd 写入 str。提醒一下，我们现在所写的 C 代码在 C 编译器眼里也是文本字符串，所以这里的“一个”转义字符\n 同样会被 C 编译器视为\和 n 两个字符。

第 183 行是在处理脚本语言中的\u，也就是 unicode 码点，这会调用 parseUnicodeCodePoint 来解析其后的 4 个十六进制数字。

如果并不是转义字符，在第 197 行直接调用 ByteBufferAdd 写入 str。

由于涉及依赖一些对象的结构，那是后几节才介绍的内容，因此现在收集到 str 中的字符在本节中没用上，待 parser 功能完整后读者就清楚了。第 200 行调用 ByteBufferClear 将 str 释放，不过不要以为此时 str 没用上就先释放它，这不是临时的做法，在将来功能完整后同样会释放此 str，毕竟释放内存还是提倡的。也许有读者会说，我看到目录中有垃圾回收，这个咱们不释放了，由垃圾回收机制来自动回收吧。这里跟读者简要说明一下，垃圾回收机制回收的是对象的内存，对象是什么呢？“直观上”是那些通过.new 方法创建的，或者是解释器在背后用创建对象方法创建的数据结构，并不回收解释器自身运行需要的内存，总之，除对象外的任何内存，垃圾回收机制概不负责，我们也是这样做的。

2.7.8 跳过注释和空行

代码中的有一些注释和空行只是对用户有用，和编译无关，因此词法分析器要去掉这些没用的部分。

stepByStep/c2/e/parser/parser.c (4/6)

```
203 // 跳过一行
204 static void skipAline(Parser* parser) {
205     getNextChar(parser);
206     while (parser->curChar != '\0') {
```

```
207         if (parser->curChar == '\n') {
208             parser->curToken.lineNo++;
209             getNextChar(parser);
210             break;
211         }
212         getNextChar(parser);
213     }
214 }
215
216 //跳过行注释或区块注释
217 static void skipComment(Parser* parser) {
218     char nextChar = lookAheadChar(parser);
219     if (parser->curChar == '/') {  // 行注释
220         skipAline(parser);
221     } else {    // 区块注释
222         while (nextChar != '*' && nextChar != '\0') {
223             getNextChar(parser);
224             if (parser->curChar == '\n') {
225                 parser->curToken.lineNo++;
226             }
227             nextChar = lookAheadChar(parser);
228         }
229         if (matchNextChar(parser, '*')) {
230             if (!matchNextChar(parser, '/')) {    //匹配*/
231                 LEX_ERROR(parser, "expect '/' after '*'!");
232             }
233             getNextChar(parser);
234         } else {
235             LEX_ERROR(parser, "expect '*/' before file end!");
236         }
237     }
238     skipBlanks(parser);  //注释之后有可能会有空白字符
239 }
240
```

第 204 行定义的 skipAline 用于跳过一行。思路是循环获取下一字符，直到遇到换行符\n，使行号自加 1 后再调用 getNextChar 获取下一字符以更新 curChar，随后退出循环。

注释有两类，以//开头的单行注释和以/*开头、以*/结尾的多行注释，多行注释也称块注释。第 217 行定义的 skipComment 用于跳过这两类注释。函数 skipComment 是在函数 getNextToken 中捕捉到字符串//或/*后调用的（等后面介绍 getNextToken 就了解了），也就是行注释和块注释都用 skipComment 处理，因此当前字符 curChar 为/或*，先记住这个结果。第 218 行调用 lookAheadChar 获得下一个字符存入变量 nextChar，nextChar 用于第 222 行的块注释部分。回来继续说 curChar，第 219 行判断 curChar 若为/，说明是行注释，那么调用 skipAline 跳过一行即可。如果是块注释，前面 nextChar 的值作为第 222 行 while 的循环条件，循环处理块注释中的换行符、更新行号，尽管这是注释，对编译器没什么用，但更新行号是为了后面代码出问题时编译器能准确报出错误所在的行。第 227 行调用 lookAheadChar 更新 nextChar 的值继续下一轮循环。

循环退出后，下一个字符要么是*，要么是源码结束的\0。第 229 行若匹配到了下一个字符是*，说明遇到了块注释的封闭部分，那么下一个字符应该是/，如果不是就报错。否则下一个字

符就是\0，这说明块注释/**/缺乏其中的*/，也就是没有封闭，因此第 235 行报错。

最后调用 skipBlanks 跳过空白符，无论是否真的有空白，由函数 skipBlanks 去处理。

2.7.9 获取 token

token 是一个单词的类型，因此获取 token 就是区别单词字符串并赋予该串类型的过程，代码如下。

stepByStep/c2/e/parser/parser.c (5/6)

```
241 //获得下一个 token
242 void getNextToken(Parser* parser) {
243     parser->preToken = parser->curToken;
244     skipBlanks(parser);  // 跳过待识别单词之前的空格
245     parser->curToken.type = TOKEN_EOF;
246     parser->curToken.length = 0;
247     parser->curToken.start = parser->nextCharPtr - 1;
248     while (parser->curChar != '\0') {
249        switch (parser->curChar) {
250           case ',':
251              parser->curToken.type = TOKEN_COMMA;
252              break;
253           case ':':
254              parser->curToken.type = TOKEN_COLON;
255              break;
256           case '(':
257              if (parser->interpolationExpectRightParenNum > 0) {
258                  parser->interpolationExpectRightParenNum++;
259              }
260              parser->curToken.type = TOKEN_LEFT_PAREN;
261              break;
262           case ')':
263              if (parser->interpolationExpectRightParenNum > 0) {
264                  parser->interpolationExpectRightParenNum--;
265                  if (parser->interpolationExpectRightParenNum == 0) {
266                      parseString(parser);
267                      break;
268                  }
269              }
270              parser->curToken.type = TOKEN_RIGHT_PAREN;
271              break;
272           case '[':
273              parser->curToken.type = TOKEN_LEFT_BRACKET;
274              break;
275           case ']':
276              parser->curToken.type = TOKEN_RIGHT_BRACKET;
277              break;
278           case '{':
279              parser->curToken.type = TOKEN_LEFT_BRACE;
280              break;
281           case '}':
```

```
            parser->curToken.type = TOKEN_RIGHT_BRACE;
            break;
        case '.':
            if (matchNextChar(parser, '.')) {
                parser->curToken.type = TOKEN_DOT_DOT;
            } else {
                parser->curToken.type = TOKEN_DOT;
            }
            break;
        case '=':
            if (matchNextChar(parser, '=')) {
                parser->curToken.type = TOKEN_EQUAL;
            } else {
                parser->curToken.type = TOKEN_ASSIGN;
            }
            break;
        case '+':
            parser->curToken.type = TOKEN_ADD;
            break;
        case '-':
            parser->curToken.type = TOKEN_SUB;
            break;
        case '*':
            parser->curToken.type = TOKEN_MUL;
            break;
        case '/':
            //跳过注释'//'或'/*'
            if (matchNextChar(parser, '/') || matchNextChar(parser, '*')) {
                skipComment(parser);

                //重置下一个 token 起始地址
                parser->curToken.start = parser->nextCharPtr - 1;

                continue;
            } else {            // '/'
                parser->curToken.type = TOKEN_DIV;
            }
            break;
        case '%':
            parser->curToken.type = TOKEN_MOD;
            break;
        case '&':
            if (matchNextChar(parser, '&')) {
                parser->curToken.type = TOKEN_LOGIC_AND;
            } else {
                parser->curToken.type = TOKEN_BIT_AND;
            }
            break;
        case '|':
            if (matchNextChar(parser, '|')) {
```

仅供非商业用途或交流学习使用




```
332                 parser->curToken.type = TOKEN_LOGIC_OR;
333             } else {
334                 parser->curToken.type = TOKEN_BIT_OR;
335             }
336             break;
337         case '~':
338             parser->curToken.type = TOKEN_BIT_NOT;
339             break;
340         case '?':
341             parser->curToken.type = TOKEN_QUESTION;
342             break;
343         case '>':
344             if (matchNextChar(parser, '=')) {
345                 parser->curToken.type = TOKEN_GREATE_EQUAL;
346             } else if (matchNextChar(parser, '>')) {
347                 parser->curToken.type = TOKEN_BIT_SHIFT_RIGHT;
348             } else {
349                 parser->curToken.type = TOKEN_GREATE;
350             }
351             break;
352         case '<':
353             if (matchNextChar(parser, '=')) {
354                 parser->curToken.type = TOKEN_LESS_EQUAL;
355             } else if (matchNextChar(parser, '<')) {
356                 parser->curToken.type = TOKEN_BIT_SHIFT_LEFT;
357             } else {
358                 parser->curToken.type = TOKEN_LESS;
359             }
360             break;
361         case '!':
362             if (matchNextChar(parser, '=')) {
363                 parser->curToken.type = TOKEN_NOT_EQUAL;
364             } else {
365                 parser->curToken.type = TOKEN_LOGIC_NOT;
366             }
367             break;
368
369         case '"':
370             parseString(parser);
371             break;
372
373         default:
374             //处理变量名及数字
375             //进入此分支的字符肯定是数字或变量名的首字符
376             //后面会调用相应函数把其余字符一并解析
377             //不过识别数字需要一些依赖，目前暂时去掉
378
379             //若首字符是字母或 "_" 则是变量名
380             if (isalpha(parser->curChar) || parser->curChar == '_') {
381                 parseId(parser, TOKEN_UNKNOWN);  //解析变量名其余的部分
```

仅供非商业用途或交流学习使用




```
382                 } else {
383                     if (parser->curChar == '#' && matchNextChar(parser, '!')) {
384                         skipAline(parser);
385                         parser->curToken.start = parser->nextCharPtr - 1;
                            //重置下一个 token 起始地址
386                         continue;
387                     }
388                     LEX_ERROR(parser, "unsupport char: \'%c\', quit.", parser->curChar);
389                 }
390                 return;
391             }
392             //大部分 case 的出口
393             parser->curToken.length = (uint32_t)(parser->nextCharPtr - parser->curToken.start);
394             getNextChar(parser);
395             return;
396         }
397 }
398
```

我们说过，语法分析器的输入是 token，每次语法分析器需要一个 token 时，它就调用词法分析器，词法分析器会返回一个 token 给它。词法分析器是一个模块，模块是一些函数的集合，语法分析器如何去调用“函数的集合”呢？肯定是有个接口，这个接口就是函数 getNextToken。函数 getNextToken 用来返回一个 token，它既是语法分析器的接口，也是整个词法分析器的入口，前面介绍过的所有函数都是为它做的准备，可以认为词法分析器就是函数 getNextToken。

第 243 行将 curToken 备份到 preToken。

接下来是初始化 curToken，token 就是单词及其周边信息的结构，curToken 就是当前匹配的单词的周边信息。第 245 行将 curToken.type 置为 TOKEN_EOF，即默认为文件结束，如果识别正常，curToken.type 会被替换为相应的 token。length 是单词的长度，第 246 行使 length 为 0，length 会在函数结束前更新为正确的值。

单词即终结符是有规则的，从一长串中匹配出符合规则的部分就是词法分析器的匹配过程，匹配只能是逐个字符判断，直到遇到“不符合规则的字符”就完成此单词的识别。就以数字的规则来说，数字不能包含字母，遇到字母时就表示匹配结束，该字母就是第一个不符合规则的字符。比如从 33a5 中匹配数字，对于词法分析器来说，数字匹配到 33 就完成了，a 是匹配单词 33 时第一个不符合规则的字符。

curChar 是当前字符，在词法分析中始终是判断 curChar 是否符合单词的规则，单词的识别过程是以“不符合规则的字符”结束，因此识别单词时如果遇到不符合规则的字符，该字符一定在 curChar 中。nextCharPtr 始终指向的是 curChar 后面的字符，每次调用 getNextToken 都会识别一个单词，getNextToken 是循环调用的，对于识别单词来说，nextCharPtr 指向的是上次调用 getNextToken 识别单词时遇到的“不符合规则的字符”的“下一个字符”的“地址”，仍以 33a5 为例，调用 getNextToken 识别出数字 33 后，curChar 为 a，nextCharPtr 是 5 的地址。总之，nextCharPtr 指向第一个不合法字符的下一个字符，因此第 247 行用 nextCharPtr 减 1 去初始化本次待识别单词的起始地址 start。

我们的词法分析器并未采用有穷状态机，而是采用按照规则“一读到底”的方式，比如若想匹配整数，那么在读到了数字字符后，继续读进接连的多个数字，直到遇到非数字为止。当然这里只是举个例子，我们不区分整数或浮点数。既然未采用有穷状态机，那么在分析单词时就不需

仅供非商业用途或交流学习使用

要逐字符判断，所以也不需要循环结构，但第 248 行的 while 循环是什么意思？

大部分情况下的“一读到底”都可以完成一个单词的识别，但这个“单词”指的是代码，即编译器感兴趣的部分，对于注释和脚本文件首行的魔数#！编译器要忽略掉。说明一下，#!用于指定解释器的路径，出现在脚本文件的首行。虽然脚本文件只是文本文件，但它在用户眼里也是程序，和二进制可执行程序是一样的，具有“能动性”的代码的化身，应该可直接执行。但它毕竟是文本文件，不能像可执行程序那样双击即可运行（或是 linux 下只要文件有执行权限，键入./文件就可执行），操作系统能执行可执行程序是因为可执行程序的格式是操作系统所认识的格式，脚本文件在操作系统眼里只被视为文本，因为它的格式就是操作系统所认识的文本格式，根本原因是脚本文件是由文本编辑器产生的，文本编辑器当然会把脚本代码存储为文本格式了。因此脚本程序必须由它的解释器去执行此文件，这是唯一的方式，脚本解释器是二进制可执行文件，操作系统可直接加载执行它。为了满足用户的需求，操作系统要求用户在脚本文件首行加入脚本解释器的路径，由操作系统“偷偷”加载解释器，把整个脚本文件作为该解释器的参数，在“背后”让解释器去执行此脚本，这样皆大欢喜。为了在脚本文件中区分解释器的路径和脚本代码，操作系统要求解释器路径必须放在首行，如果用户就想显式用“解释器 脚本文件”的方式运行，那么首行有可能就是普通的脚本代码，因此操作系统规定：为防止这种情况，如果首行是解释器路径的话，那么就在路径前加个字符串#!。即首行以#!开头的字符串被操作系统认为是解释器路径。#!之所以称之为魔数，是因为只有脚本和操作系统程序加载器才知道#!的意思，很神秘，因此称为魔数。也许有读者会说，这分明是魔“字符串”，哪有“数”啊，其实最开始的魔数是完全的数字，比如主引导程序最后两个字节是 0x55aa，这就是魔数，魔数是从一开始传下来的，因此沿用至今。可是如果脚本文件首行有魔数#!，但又采用“解释器 脚本文件”的方式执行怎么办？也就是说让脚本文件作为解释器的参数去执行，那么脚本解释器会不会把#1 所在的首行当成代码？一个健壮的解释器就应该检测到这种情况，应该跳过#!所在的行，这也是我们需要跳过#！的原因。每调用一次 getNextToken 便要获得一个 token，在跳过代码中的注释或魔数#!所在的整行后，任务还没有完成，必须得识别一个 token 才能返回，因此这种情况下需要回到识别的起始点，以便继续识别下一个单词完成任务，还是循环结构比较合适。后面你会看到，在需要跳过注释或#!的后面都用了 continue 语句回到 while 循环的开始。

提前说明一下，token 中还有一个 length 成员，它是单词的长度，我们在循环体的最后即 switch 之后做统一的处理。

在第 250 行，如果 curChar 是逗号“,”，那么将 curToken.type 置为 TOKEN_COMMA，跳出 switch，到循环体结束处设置单词,的 length，完成本次识别。对于这种单字符的单词，都采用这样的处理逻辑。

第 256 行处理左小括号“(”，由于“(”也是内嵌表达式的一部分，这里还要跟踪嵌套层数，前面说过了，我们只支持一层嵌套。在第 257 行判断如果 interpolationExpectRightParenNum 大于 0，说明之前已经遇到了内嵌表达式% “(”，因此第 258 行将其自加 1。当下次再遇到%(时用来判断嵌套的层数。第 260 行将 curToken.type 置为 TOKEN_LEFT_PAREN，退出循环完成本次识别。

第 262 行处理右小括号 “)”，它依然与内嵌表达式有关。如果 interpolationExpectRightParenNum 的值大于 0，说明之前曾遇到过% “(”，此时期望的右小括号 “)” 出现了，因此使 interpolationExpectRightParenNum 减 1。减 1 后若为 0，说明没有嵌套，在第 266 行调用 parseString 去解析内嵌表达式所在字符串的尾部的字符串，比如 abc%(9)def 中的 def，前面说过了此部分被视为字符串，在 parseString 中会设置 curToken.type 为 TOKEN_STRING。设置完 token 类别后就完成匹配了，因此第 267 行跳出 switch，到循环体末尾处设置字符串的 length。如果 interpolationExpectRightParenNum 的值并不大

仅供非商业用途或交流学习使用



于 0，说明之前没有内嵌表达式，因此直接设置 token 类型为 TOKEN_RIGHT_PAREN，退出 switch，跳到循环体结尾处设置单词“)”的 length。

第 284～290 行用于处理“..”和“.”。“..”用于表示 range，如 1..5 表示 1 2 3 4 5 这 5 个数字。“.”表示方法调用，即“对象“.”方法”中的“.”。当识别到“.”时，调用 matchNextChar 判断下一个字符是不是“.”，如果是，说明识别到了“..”，因此置 token 为 TOKEN_DOT_DOT，否则就是 TOKEN_DOT。

第 291～297 行判断识别到的是==还是=，原理同上，不解释。

第 307～319 行处理除号/和注释，注释包括块注释/*和行注释//，如果是注释就调用函数 skipComment 处理，然后执行 continue 跳到循环开始，继续识别单词。否则就是除号，curToken.type 设置为 TOKEN_DIV。

default 中即第 380 行用于处理变量名及数字，变量名首字符是字母或下划线_，第 380 行判断若是这种情况，在第 381 行调用 parseId 去解析变量名“其余的”部分，说“其余”是因为当前已经读了标识符的首字符。不过，数字识别后，其存储涉及对象中一个结构，等到完成对象时再补充。

第 383 行识别到魔数#!，第 384 行调用 skipAline 跳过所在的行。

第 393 行用于统一处理单词的 length，其值是 nextCharPtr 与 curToken.start 的差。

2.7.10　token 匹配和初始化 parser

如题，还有初始化函数及一些供语法分析器调用的函数，代码如下。

stepByStep/c2/e/parser/parser.c (6/6)

```
399 //若当前 token 为 expected 则读入下一个 token 并返回 true,
400 //否则不读入 token 且返回 false
401 bool matchToken(Parser* parser, TokenType expected) {
402     if (parser->curToken.type == expected) {
403         getNextToken(parser);
404         return true;
405     }
406     return false;
407 }
408
409 //断言当前 token 为 expected 并读入下一 token，否则报错 errMsg
410 void consumeCurToken(Parser* parser, TokenType expected, const char* errMsg) {
411     if (parser->curToken.type != expected) {
412         COMPILE_ERROR(parser, errMsg);
413     }
414     getNextToken(parser);
415 }
416
417 //断言下一个 token 为 expected,否则报错 errMsg
418 void consumeNextToken(Parser* parser, TokenType expected, const char* errMsg) {
419     getNextToken(parser);
420     if (parser->curToken.type != expected) {
421         COMPILE_ERROR(parser, errMsg);
422     }
423 }
424
```

仅供非商业用途或交流学习使用


```
425 //由于 sourceCode 未必来自于文件 file,有可能只是个字符串,
426 //file 仅用作跟踪待编译的代码的标识,方便报错
427 void initParser(VM* vm, Parser* parser, const char* file, const char* sourceCode) {
428     parser->file = file;
429     parser->sourceCode = sourceCode;
430     parser->curChar = *parser->sourceCode;
431     parser->nextCharPtr = parser->sourceCode + 1;
432     parser->curToken.lineNo = 1;
433     parser->curToken.type = TOKEN_UNKNOWN;
434     parser->curToken.start = NULL;
435     parser->curToken.length = 0;
436     parser->preToken = parser->curToken;
437     parser->interpolationExpectRightParenNum = 0;
438     parser->vm = vm;
439 }
```

第 401 行定义的函数 matchToken 用于匹配当前 token 是否为参数 expected，如果是就调用 getNextToken 读入下一个 token 并返回 true，否则返回 false。

第 410 行定义的 consumeCurToken 是一种断言，即断定当前 token 一定是 expected，如果不是就调用 COMPILE_ERROR 报编译型错误，报错信息是 errMsg。consume 所指的“消费”是指通过读入下一个 token 的方式换掉当前 token，即消费掉当前的 token。

第 418 行定义的 consumeNextToken 用于断言下一个 token 为 expected，否则报错 errMsg。既然是断言下一个 token，因此第 419 行先调用 getNextToken 获得下一个 token 换掉 curToken。如果类型不符合 expected 就报错。

第 427 行定义的 initParser 用于初始化词法分析器，file 通常情况下是所编译的源码文件名，sourceCode 是源码串。初始化情况下 curToken.type 设置为 TOKEN_UNKNOWN，待识别完成会被更正为相应类型。重点说下第 430 行和第 431 行，这是词法分析器更有效率的地方。这两行代码使词法分析器的当前字符 curChar 等于源码串的首字符，并使下一个字符的地址 nextCharPtr 等于源码串首字符的下一个字符，这样在获取下一字符时可以直接从源码串中获取，每次读取一个字符后再使 nextCharPtr 自加 1 就完成了更新，无须调用 getChar 之类的函数，避免了系统调用的开销。

2.8 构建主程序（本节源码 stepByStep/c2/f）

词法分析器需要一个命令行入口，它也是我们的主程序，主程序有两个功能，一是负责编译并运行脚本文件，二是负责解析命令行，也就是实时编译。

本节建立了一个新目录：cli，主程序文件是该目录下的 cli.c。代码如下所示。

stepByStep/c2/f/cli/cli.c

```
1 #include "cli.h"
2 #include <stdio.h>
3 #include <string.h>
4 #include "parser.h"
5 #include "vm.h"
6 #include "core.h"
7
8 //执行脚本文件
```

仅供非商业用途或交流学习使用


```
 9 static void runFile(const char* path) {
10     const char* lastSlash = strrchr(path, '/');
11     if (lastSlash != NULL) {
12         char* root = (char*)malloc(lastSlash - path + 2);
13         memcpy(root, path, lastSlash - path + 1);
14         root[lastSlash - path + 1] = '\0';
15         rootDir = root;
16     }
17
18     VM* vm = newVM();
19     const char* sourceCode = readFile(path);
20
21     struct parser parser;
22     initParser(vm, &parser, path, sourceCode);
23
24     #include "token.list"
25     while (parser.curToken.type != TOKEN_EOF) {
26        getNextToken(&parser);
27        printf("%dL: %s [", parser.curToken.lineNo, tokenArray[parser.curToken.type]];
28        uint32_t idx = 0;
29        while (idx < parser.curToken.length) {
30           printf("%c", *(parser.curToken.start+idx++));
31        }
32        printf("]\n");
33     }
34 }
35
36 int main(int argc, const char** argv) {
37     if (argc == 1) {
38         ;
39     } else {
40         runFile(argv[1]);
41     }
42     return 0;
43 }
```

第 9 行定义的 runFile 用于执行脚本文件，参数 path 是文件的路径。执行方式如同 python file.py，其中的 file.py 就是脚本文件。

第 10～16 行用于设置 rootDir。它是脚本文件所在的根目录，设置此目录的目的是为了找到脚本文件，脚本文件若在当前目录下则很容易找到，无须设置 rootDir，所以，如果脚本文件并不在当前路径才需要设置 rootDir。

第 10 行用于判断 path 是否为当前路径的形式，如果包含/，说明“可能”并不是在当前目录下（./形式的路径也是当前路径，因此说“可能”，不过这不影响路径的正确性），为保存路径信息申请动态内存，用 root 指向该内存地址，将路径复制到该内存后，用全局变量 rootDir 指向该内存地址。前面说过，rootDir 是定义在 core.c 中的全局变量，表示脚本文件的根目录。如果 path 中并不包含/，说明是当前路径，因此不需要设置 rootDir。

第 18 行新建一虚拟机，用于运行脚本文件或实时解释执行从命令行中读取的代码。第 19 行调用 readFile 获取文件的源码到 sourceCode。第 22 行调用 initParser 初始化新建的 parser。

仅供非商业用途或交流学习使用

以下都是为演示词法分析器临时创建的，第 24 行包含的 token.list 也是为此临时创建的，其内容就是一个 token 字符串数组 tokenArray，此数组用于打印所识别到的 token 的名称。稍后会贴出该数组的定义。

第 25～33 行用于循环打印词法分析器对脚本源码的识别结果，第 27 行打印行号及 token 的字符串形式，第 29 行的 while 用于打印 token 对应的源码串。一会见执行结果就清晰了。

第 36 行就是主函数了，第 37 行是为将来实现命令行预留的，如果 argc 为 1，说明用户只执行了解释器，并未指定脚本文件，只想运行命令行，于是应该去调用执行命令行的函数，将来会完成。第 40 行是编译运行脚本文件，我们前期先完成这个功能。

刚才说到的 tolen.list 临时先放到 parser 目录下了，如下所示。

stepByStep/c2/f/parser/token.list

```
char* tokenArray[] = {
    "UNKNOWN",
    "NUM",
    "STRING",
    "ID",
    "INTERPOLATION",
    "VAR",
    "FUN",
    "IF",
    "ELSE",
    "TRUE",
    "FALSE",
    "WHILE",
    "FOR",
    "BREAK",
    "CONTINUE",
    "RETURN",
    "NULL",
    "CLASS",
    "THIS",
    "STATIC",
    "IS",
    "SUPER",
    "IMPORT",
    "COMMA",
    "COLON",
    "LEFT_PAREN",
    "RIGHT_PAREN",
    "LEFT_BRACKET",
    "RIGHT_BRACKET",
    "LEFT_BRACE",
    "RIGHT_BRACE",
    "DOT",
    "DOT_DOT",
    "ADD",
    "SUB",
    "MUL",
```

仅供非商业用途或交流学习使用


```
    "DIV",
    "MOD",
    "ASSIGN",
    "BIT_AND",
    "BIT_OR",
    "BIT_NOT",
    "BIT_SHIFT_RIGHT",
    "BIT_SHIFT_LEFT",
    "LOGIC_AND",
    "LOGIC_OR",
    "LOGIC_NOT",
    "EQUAL",
    "NOT_EQUAL",
    "GREATE",
    "GREATE_EQUAL",
    "LESS",
    "LESS_EQUAL",
    "QUESTION",
    "EOF"
};
```

2.9 编译、测试（本节源码 stepByStep/c2/f）

2.9.1 一个简单的 makefile

现在主函数也有了，应该开始编译了，为免去将来一遍遍的手工编译，现在就写个 makefile 吧，简单够用就行，如下所示。

stepByStep/c2/f/makefile

```
1 CC = gcc
2 #CFLAGS = -g -DDEBUG -lm -Wall -I object -I vm -I compiler -I parser -I include -I
  cli -I gc -W -Wstrict-prototypes -Wmissing-prot otypes -Wsystem-headers
3 CFLAGS = -g -lm -Wall -I object -I vm -I compiler -I parser -I include -I cli -I gc
  -W -Wstrict-prototypes -Wmissing-prototypes -W system-headers
4 TARGET = spr
5 DIRS =  object include cli compiler parser vm gc
6 CFILES = $(foreach dir, $(DIRS),$(wildcard $(dir)/*.c))
7 OBJS = $(patsubst %.c,%.o,$(CFILES))
8 $(TARGET):$(OBJS)
9         $(CC) -o $(TARGET) $(OBJS) $(CFLAGS)
10 clean:
11         -$(RM) $(TARGET) $(OBJS)
12 r: clean $(TARGET)
```

makefile 并不属于编译器的内容，这里只大概说明，我们这里本着够用就行的原则，详细文档网上很多，读者感兴趣就自行参阅吧。

如果初次接触 makefile，可能会误以为它是用来编译程序的，其实并不完全是这样，编译程序的幕后功臣是 makefile 中指定的编译器，在上述 makefile 中指的是 gcc。编写 makefile 其实是

仅供非商业用途或交流学习使用

在编程，就像写 shell 脚本一样。makefile 也是文本文件，因此它的执行也要有解释器才行，此解释器就是 make 程序，make 之所以能编译程序，是因为它调用了 gcc，make 只是按照 makefile 的意思执行而已，makefile 中不包括和编译相关的命令也无妨，只是 makefile 的主要用途是编译而已。

这个 makefile 会把目录 object、vm、compiler、parser、include、cli 和 gc 中的.c 文件编译为可执行程序 spr（sparrow），目前只有 vm、parser 和 include 三个目录，为了省事，现一次把未来创建的目录也写进去了，这次写完后就不需要修改了。说明一下，尽管此时我们还没有完成那么多目录，但 makefile 会自动略过那些不存在的目录，因此并不报错。下面逐行介绍。

第 1 行定义的变量 CC 用来指定编译器，这里为其赋值为 gcc。

第 2 行定义的 CFLAGS 是 gcc 的参数变量，其中，

-g 用于指定 gcc 包含符号表信息，用于调试。

-D 用于“自定义”一个宏，并不是指定某个标准宏名。这里的宏名是 DEBUG，给我们调试用的，相当于在我们的 C 源码中定义的一样，在源码中对宏 DEBUG 做判断，如果已经定义了 DEBUG 就采用另一套调试代码。一般情况下在参数中定义宏是为了不修改 C 源码，这是较基本的调试方式，一个好的调试方式不建议总是修改源码。

-l（小写的 L）用于编译时指定包含一个依赖库，其后的 m 表示数学库 math。

-I（大写的 i）用于指定依赖文件所在的目录，比如头文件，GCC 会在用-I 指定的目录中找头文件，这里用-I 分别指定我们自己的目录。

-W、-Wstrict-prototypes、-Wmissing-prototypes 和-Wsystem-headers 用于指定编译时的报警级别。

目前尚未到调试阶段，因此在 CFLAGS 前加#注掉，先用第 3 行的 CFLAGS。

第 4 行的变量 TARGET 的值是 spr，我们用 TARGET 作为编译结果。

第 5 行的变量 DIRS 用于指定待编译文件所在的目录集合，也就是要编译哪些目录下的 C 文件。

make 提供了一些内建函数或命令，比如 foreach 用于迭代数组变量中的每个值，wildcard 是通配符，“通用匹配”一些共性的东西，patsubst 用于替换字符串，结果引用要用$()括起来。

第 6 行的代码 CFILES =$(foreach dir, $(DIRS),$(wildcard $(dir)/*.c))就是迭代 DIRS 变量（前面说过，它是所有待编译文件所在的目录的集合，定义在第 5 行）中的每个目录下的.c 文件，迭代的结果存储到变量 CFILES 中，因此变量 CFILES 是所有待编译的文件的集合。

第 7 行的 OBJS = $(patsubst %.c,%.o,$(CFILES))就是把变量 CFILES 中所有.c 替换为.o 后存储到变量 OBJS，因此变量 OBJS 是所有.o 目标文件的集合。

makefile 的执行方式是目标依赖，递归执行，这里的目标并不是专指.o 形式的目标文件，而是指规则中的目标，此目标泛指一切待构建的对象。规则是一组用：区分的目标和依赖，然后下一行是以 tab 开头的“动作”，格式是

```
目标：依赖 1 依赖 2 … 依赖 n
“\t” 命令
```

目标只有一个，依赖可以有多个，依赖之间用空格符分隔，如果目标没有依赖的话，那么目标就称为伪目标（后面介绍），因为真正需要构建的目标不会没有依赖。因此但凡没有依赖的目标，就说明该目标并不需要“真正”地去构建，被称为伪目标。目标可以是可执行文件，也可以是.o 文件，甚至可以是伪目标。规则表达的是，要想构建目标，先要完成目标所依赖的部分，如果依

赖尚未存在，那么先去构建依赖，如果此依赖又是另一规则的目标，继续递归构造，因此是属于自上而下的运行方式。

第 8 行的$(TARGET):$(OBJS)表示变量 TARGET 依赖变量 OBJS，当依赖的 OBJS 存在后，执行第 9 行以 tab 开头的命令串$(CC) -o $(TARGET) $(OBJS) $(CFLAGS)，这个就是编译的命令。执行后会生成可执行程序 spr。

第 10 行的 clean：是个伪目标，伪目标是不需要实际构造的对象，这和 make 的使命是相悖的，那么它存在的意义是什么呢？简单说下缘由。尽管构建目标是 make 的任务，但 make 为了不做无用功，它会判断构建目标的那些依赖是否有过变更，有过变更再重新构造目标才会有意义，否则依赖都没变化，那么目标也肯定是最新的，即使重新构建也依然会和之前的一样，因此就不执行创建目标的命令（即以 tab 开头的命令串）了。也就是说，如果想每次执行 make 时都会执行规则中的命令，就要确保依赖有更新，难道为了执行命令要手工修改一下文件？这代价太大了。如果目标没有依赖呢？那么 make 就没法确定目标是不是最新的。因此 make 直接执行构建目标的命令伪目标就成了单纯的执行命令。clean 的命令是第 11 行的-$(RM) $(TARGET) $(OBJS)，也就是调用 rm 命令删除可执行文件 spr 和所有的.o 文件。这里注意，在$(RM)左边有一个-，它表示忽略后面命令的警告。这里的命令是 rm，如果用 rm 删除的文件即 rm 的参数不存在，rm 会报警，这里忽略它的报警信息不输出。

补充说明，make 判断依赖是否有更新的方法是，检查依赖文件的 mtime 是否比目标文件的 mtime 更新。正常情况下 make 会自动检查目标的依赖文件是否有更新，这样在执行 make 时会判断是否重建目标。随着工程越来越大，目标依赖的文件越来越多，这需要把所有依赖都显式放在规则中的目标后。但手工查找所有的依赖文件太累了，为此 gcc 提供-MM 参数来搜索所有的依赖文件（不包括标准库），直接输出规则，冒号左边是目标，冒号右边是空格分隔的所有依赖文件，如图 2-3 所示。

```
[work@work f]$ gcc -MM vm/vm.c -I vm -I parser -I include -I cli
vm.o: vm/vm.c vm/vm.h include/common.h include/utils.h include/common.h
[work@work f]$
```

图 2-3

读者可以把输出直接写在 makefile 中。gcc 还提供了-M 参数，与-MM 的区别是，它会把标准库的依赖也列出来，也就是跟踪在#include 后尖括号<>中的标准库头文件，如图 2-4 所示。

```
[work@work f]$ gcc -M vm/vm.c -I vm -I parser -I include -I cli
vm.o: vm/vm.c vm/vm.h include/common.h /usr/include/stdio.h \
 /usr/include/features.h /usr/include/sys/cdefs.h \
 /usr/include/bits/wordsize.h /usr/include/gnu/stubs.h \
 /usr/include/gnu/stubs-64.h \
 /usr/lib/gcc/x86_64-redhat-linux/4.4.7/include/stddef.h \
 /usr/include/bits/types.h /usr/include/bits/typesizes.h \
 /usr/include/libio.h /usr/include/_G_config.h /usr/include/wchar.h \
 /usr/lib/gcc/x86_64-redhat-linux/4.4.7/include/stdarg.h \
 /usr/include/bits/stdio_lim.h /usr/include/bits/sys_errlist.h \
 /usr/include/stdlib.h /usr/include/bits/waitflags.h \
 /usr/include/bits/waitstatus.h /usr/include/endian.h \
 /usr/include/bits/endian.h /usr/include/bits/byteswap.h \
 /usr/include/sys/types.h /usr/include/time.h /usr/include/sys/select.h
 /usr/include/bits/select.h /usr/include/bits/sigset.h \
 /usr/include/bits/time.h /usr/include/sys/sysmacros.h \
 /usr/include/bits/pthreadtypes.h /usr/include/alloca.h \
 /usr/include/stdint.h /usr/include/bits/wchar.h include/utils.h \
 include/common.h
[work@work f]$
```

图 2-4

但还是很麻烦，我在目标中并未明确列出所有的依赖文件，因为每次有新文件产生时都要手工再执行一次依赖排查，因此有可能某些文件在被更新后，执行 make 时 make 会提示没有更新，也就是 make 无效。为了省事，干脆把目标文件全删重建，也就是先执行 make clean，再执行 make，但这样还得执行两次，最好是合成一步，这就是下行规则的作用。

r: clean $(TARGET)用于先执行目标 clean，再执行目标$(TARGET)，这是利用了 makefile 的依赖关系，r 在此也是目标，是自定义的，在此表示 rebuild，r 依赖的 clean 要先执行，然后再执行目标$(TARGET)，因此实现了 rebuild 功能。

makefile 写好了，下面执行 make，输出如下。

```
[work@work f]$ make
gcc -g -lm -Wall -I object -I vm -I compiler -I parser -I include -I cli -I gc -W
-Wstrict-prototypes -Wmissing-prototypes -Wsystem-headers   -c -o include/unicodeUtf8.o
include/unicodeUtf8.c
gcc -g -lm -Wall -I object -I vm -I compiler -I parser -I include -I cli -I gc -W
-Wstrict-prototypes  -Wmissing-prototypes  -Wsystem-headers   -c  -o  include/utils.o
include/utils.c
gcc -g -lm -Wall -I object -I vm -I compiler -I parser -I include -I cli -I gc -W
-Wstrict-prototypes -Wmissing-prototypes -Wsystem-headers   -c -o cli/cli.o cli/cli.c
gcc -g -lm -Wall -I object -I vm -I compiler -I parser -I include -I cli -I gc -W
-Wstrict-prototypes  -Wmissing-prototypes  -Wsystem-headers   -c  -o  parser/parser.o
parser/parser.c
gcc -g -lm -Wall -I object -I vm -I compiler -I parser -I include -I cli -I gc -W
-Wstrict-prototypes -Wmissing-prototypes -Wsystem-headers   -c -o vm/core.o vm/core.c
gcc -g -lm -Wall -I object -I vm -I compiler -I parser -I include -I cli -I gc -W
-Wstrict-prototypes -Wmissing-prototypes -Wsystem-headers   -c -o vm/vm.o vm/vm.c
gcc -o spr include/unicodeUtf8.o include/utils.o cli/cli.o parser/parser.o vm/core.o
vm/vm.o  -g -lm -Wall -I object -I vm -I compiler -I parser -I include -I cli -I gc -W
-Wstrict-prototypes -Wmissing-prototypes -Wsystem-headers
[work@work f]$
```

make 过后已在各个目录下生成了.o 文件，在最外层目录生成了可执行文件 spr。执行 ls *查看下结果。

```
[work@work f]$ ls *
makefile  sample.sp  spr

cli:
cli.c  cli.h  cli.o

include:
common.h  unicodeUtf8.c  unicodeUtf8.h  unicodeUtf8.o  utils.c  utils.h  utils.o

parser:
parser.c  parser.h  parser.o  token.list

vm:
core.c  core.h  core.o  vm.c  vm.h  vm.o
[work@work f]$
```

下面再测试下 clean 的功能，如下所示。

```
[work@work f]$ make clean
rm -f spr include/unicodeUtf8.o include/utils.o cli/cli.o parser/parser.o vm/core.o
vm/vm.o
```

伪目标 clean 就是执行了 rm –f 的命令，ls 查看下是否真的删除了。

```
[work@work f]$ ls *
makefile  sample.sp

cli:
cli.c  cli.h

include:
common.h  unicodeUtf8.c  unicodeUtf8.h  utils.c  utils.h

parser:
parser.c  parser.h  token.list

vm:
core.c  core.h  vm.c  vm.h
[work@work f]$
```

2.9.2　测试 paser

本小节测试 parser 识别单词的功能，为此写了一个简单的测试文件，此文件便是 sparrow 脚本代码，注意，由于目前尚未支持数字的识别，所以样本中不能有数字，内容如下所示。

stepByStep/c2/f/ sample.sp

```
 1 import people for People
 2 fun fn() {
 3     var p = People.new("xiaoming", "male")
 4     p.sayHi()
 5 }
 6
 7 class Family < People {
 8     var father
 9     var mother
10     var child
11     new(f, m, c) {
12        father = f
13        mother = m
14        child  = c
15        super("wbf", "male")
16     }
17 }
18
19 var f = Family.new("wbf", "ls", "shine")
20 f.sayHi()
21
22 fn()
```

按照下述方式执行，输出如下。

```
[work@work f]$ ./spr sample.sp
1L: IMPORT [import]
1L: ID [people]
1L: FOR [for]
1L: ID [People]
2L: FUN [fun]
2L: ID [fn]
2L: LEFT_PAREN [(]
2L: RIGHT_PAREN [)]
2L: LEFT_BRACE [{]
3L: VAR [var]
3L: ID [p]
3L: ASSIGN [=]
3L: ID [People]
3L: DOT [.]
3L: ID [new]
3L: LEFT_PAREN [(]
3L: STRING ["xiaoming"]
3L: COMMA [,]
3L: STRING ["male"]
3L: RIGHT_PAREN [)]
4L: ID [p]
4L: DOT [.]
4L: ID [sayHi]
4L: LEFT_PAREN [(]
4L: RIGHT_PAREN [)]
5L: RIGHT_BRACE [}]
7L: CLASS [class]
7L: ID [Family]
7L: LESS [<]
7L: ID [People]
7L: LEFT_BRACE [{]
8L: VAR [var]
8L: ID [father]
9L: VAR [var]
9L: ID [mother]
10L: VAR [var]
10L: ID [child]
11L: ID [new]
11L: LEFT_PAREN [(]
11L: ID [f]
11L: COMMA [,]
11L: ID [m]
11L: COMMA [,]
11L: ID [c]
11L: RIGHT_PAREN [)]
11L: LEFT_BRACE [{]
12L: ID [father]
```




```
12L: ASSIGN [=]
12L: ID [f]
13L: ID [mother]
13L: ASSIGN [=]
13L: ID [m]
14L: ID [child]
14L: ASSIGN [=]
14L: ID [c]
15L: SUPER [super]
15L: LEFT_PAREN [(]
15L: STRING ["wbf"]
15L: COMMA [,]
15L: STRING ["male"]
15L: RIGHT_PAREN [)]
16L: RIGHT_BRACE [}]
17L: RIGHT_BRACE [}]
19L: VAR [var]
19L: ID [f]
19L: ASSIGN [=]
19L: ID [Family]
19L: DOT [.]
19L: ID [new]
19L: LEFT_PAREN [(]
19L: STRING ["wbf"]
19L: COMMA [,]
19L: STRING ["ls"]
19L: COMMA [,]
19L: STRING ["shine"]
19L: RIGHT_PAREN [)]
20L: ID [f]
20L: DOT [.]
20L: ID [sayHi]
20L: LEFT_PAREN [(]
20L: RIGHT_PAREN [)]
22L: ID [fn]
22L: LEFT_PAREN [(]
22L: RIGHT_PAREN [)]
23L: EOF []
[work@work f]$
```

目测功能符合预期，不过目前的 parser 功能尚不完整，等第 3 章实现对象等相关结构后再完善。



第3章　类与对象

在完善词法分析器之前，必须先实现类和对象。对象和类是面向对象语言中必不可少的部分，在 sparrow 中，一切皆对象。对象是类的实例化，为在程序中区分这两种概念，约定与对象相关的命名，如文件和标识符，为显式表示其是对象而不是类，都会包含关键字 obj。

3.1 对象在 C 语言中的概貌

在面向对象编程中，对象是具有特定属性和行为的实体，对象的类型就是类。对象是类的具体实例，类是对象的模板，表示一类对象的共性特征。任何对象都有它所属的类，也就是任何对象都有它的类型。

我们在实现一门面向对象的语言，因此必然要实现一些与对象相关的结构。我们的 parser 在处理数字和字符串时会依赖这些结构，所以先完成这些基础构件。

你是否好奇如何用面向过程的 C 语言表示一个“对象”？“类”是一类具有共性事物的抽象，共性往往具有通用性，类是这些通用数据的模板，数据包括方法（method）和域（field），域也称为字段或属性。对象是类这个通用模板的实例化，对象彼此之间是独立、隔离的，如何维护各个对象自己的数据？本小节先来个热身。

为陈述清楚类和对象，现在需要达成一致，这里把在 C++或 Java 中用 class 关键字定义的数据类型称为类，也称 class。通过调用类的构造函数创建的类的实例称为对象，也称 object。其实这就是我们理解的类和对象，对象是类的实例化。

我们是用 C 语言来实现对象，那么自然要用 C 中的数据结构，对象在外观上很像 C 语言中的结构体，而且结构体是 C 语言的核心，理所当然地用结构体维护对象的数据。我们是用一个名为 class 的 struct 结构构建类，用一个名为 ObjInstance 的 struct 结构构建对象（但并不是所有对象都要用到此结构），为了跟踪各个对象所属的类，每个对象都有一个通用的结构——对象头，在对象头中包含了对象类型和对象所属的类，同时，为了跟踪系统中所有已分配的对象，我们在对象头中加入另外一个成员——对象头指针，用于指向下一个对象的对象头，这样就构成了“所有已分配对象”链表，将来实现的垃圾回收 GC 就会用到此链表。说明一下，这里说的是链接所有已分配的对象，已分配的对象是从堆里分配的，也就是通过 malloc 或 new 函数分配的内存所创建的对象，并不单指本类的所有对象。值得说明的是，class 也有其所属的类——meta 类，即类的类（此时你可能觉得奇怪，其实是合理的，原因以后我们会详述），因此在 class 中也要有对象头。

前面说过，类是通用数据的模板，因此一切通用的部分都放在 class 中。对象之间虽然是彼此隔离的，但它们所拥有的方法都是一样的，因此方法放在类中共享，我们这里是用 methods 数组来存储类中所有方法。类也有自己的基类，因此 class 中也要有成员去指向其基类，这里是用名为 superClass 的指针指向基类。对象自己的域放在对象自己的 ObjInstance 结构中的 fields 数组中。大致情形如图 3-1 所示。



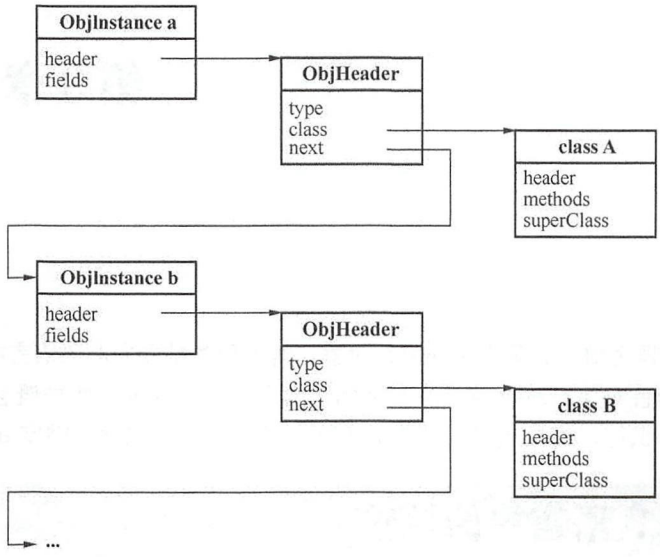


图 3-1

说明一下，图 3-1 中对象和类中的 header 并不是指针，它们就是 ObjHeader 结构。ObjHeader 中的 class 和 next 是指针。

对于陌生领域的认识，都是先利用已了解的事物去想象，根据已知去想象，理解未知，如果本小节内容无法让你展开想象，一种可能的原因是缺乏那些已知的基础，帮你建立这些基础正是本书的使命，请你在后面实践中慢慢体会。

3.2 实现对象头（本节源码 stepByStep/c3/a）

就像邮件一样，邮件头是邮件的元信息，对象头是对象的元信息，相当于对象在语言系统中的身份证，我们通过对象头来组织、跟踪、辨别所有的对象类型。它是各个对象结构和类结构中的核心成员及首成员。

本节建立目录 object，对象头的头文件是 object/header_obj.h，代码如下。

stepByStep/c3/a/object/header_obj.h

```
#ifndef _OBJECT_HEADER_H
#define _OBJECT_HEADER_H
#include "utils.h"
typedef enum {
    OT_CLASS,    //此项是 class 类型，以下都是 object 类型
    OT_LIST,
    OT_MAP,
    OT_MODULE,
    OT_RANGE,
    OT_STRING,
    OT_UPVALUE,
    OT_FUNCTION,
    OT_CLOSURE,
    OT_INSTANCE,
    OT_THREAD
```

```
16 } ObjType;  //对象类型
17
18 typedef struct objHeader {
19     ObjType type;
20     bool isDark;     //对象是否可达
21     Class* class;    //对象所属的类
22     struct objHeader* next;    //用于链接所有已分配对象
23 } ObjHeader;       //对象头，用于记录元信息和垃圾回收
24
25 typedef enum {
26     VT_UNDEFINED,
27     VT_NULL,
28     VT_FALSE,
29     VT_TRUE,
30     VT_NUM,
31     VT_OBJ    //值为对象,指向对象头
32 } ValueType;      //value 类型
33
34 typedef struct {
35     ValueType type;
36     union {
37         double num;
38         ObjHeader* objHeader;
39     };
40 } Value;    //通用的值结构
41
42 DECLARE_BUFFER_TYPE(Value)
43
44 void initObjHeader(VM* vm, ObjHeader* objHeader, ObjType objType, Class* class);
45 #endif
```

第 4 行定义的 ObjType 是对象类型。

OT_CLASS 表示 class 类型，即类。

OT_LIST 是 list 类型，即列表。

OT_MAP 是 map 类型，即散列数组。

OT_MODULE 是 module 类型，即模块作用域。

OT_RANGE 是 range 类型，即表示步进为 1 的数字范围。

OT_STRING 是 string 类型，即字符串。

OT_UPVALUE 是 upvalue 类型，即 upvalue（上值），其实就是自由变量，这是闭包中的概念，第 0 章中提到过，后面实现时再细说。

OT_FUNCTION 是 function 类型，即函数。

OT_CLOSURE 是 closure 类型，即闭包，第 0 章介绍了闭包。

OT_INSTANCE 是 instance，即实例，对象实例。

OT_THREAD 是 thread，即线程类型。

第 18 行定义的 ObjHeader 即是对象头，在面向对象脚本语言中是万物的基础。Type 指明对象类型，isDark 表示对象是否可达，这是将来用于垃圾回收，先放这儿。指针 class 指向本对象所属的类，其类型是 Class，下节会介绍其定义。指针 next 指向下一个创建的对象，所有对象链接



形成了已分配对象链表。

第 25 行定义的 ValueType 是值类型，脚本语言对用户是弱类型，但既然是用强类型的 C 语言来实现，我们需要在背后区分数据类型以分别对待。VT_UNDEFINED 表示未定义，VT_NULL 表示空值，空值并不是未定义，之前容易把两类混淆。VT_FALSE 表示布尔假，VT_TRUE 表示布尔真。VT_NUM 表示数字类型，VT_OBJ 表示值为一个对象。

第 34 行定义的 Value 就是通用的值结构，type 表示此值的类型，第 36 行的 union 表示数值，包括数字和对象两种。为通用指向所有类型的对象，第 38 行是用对象头来表示的，因此值为对象时，用 objHeader 指向具体对象的 header。

第 42 行是用宏 DECLARE_BUFFER_TYPE(Value)生成 Value 类型 buffer 的定义，即 ValueBuffer 及与 ValueBuffer 配套的 4 个操作方法的声明，即 VaLueBufferInit、VaLueBufferFillWrite、VaLueBufferAdd 和 VaLueBufferClear 的声明。前面介绍过此宏，此宏须与宏 DEFINE_BUFFER_METHOD（用于定义方法）成套使用以生成 Valuebuffer 操作函数的定义，在 header_obj.c 中会调用。

第 44 行是函数 initObjHeader 的声明，其定义在文件 header_obj.c 中。

在继续之前还要在虚拟机中新增一个成员，以后也会经常变更。另外，除了包含了新的头文件和头文件中的函数声明外，源码中有任何变更我都会在书上贴出来，保证能在书上找到出处。

stepByStep/c3/a/vm/vm.h

```
…略
 7 struct vm {
 8     Class* stringClass;
 9     uint32_t allocatedBytes; //累计已分配的内存量
10     ObjHeader* allObjects;   //所有已分配对象链表
11     Parser* curParser;       //当前词法分析器
12 };
…略
```

第 10 行的 allObjects 是新增的，这是所有已分配对象链表的首节点，也就是说我们在 VM 中跟踪所有已分配对象。

新增 allObjects 后，还要在 initVM 中新增对其初始化。

stepByStep/c3/a/vm/vm.c

```
…略
 4 //初始化虚拟机
 5 void initVM(VM* vm) {
 6     vm->allocatedBytes = 0;
 7     vm->allObjects = NULL;
 8     vm->curParser = NULL;
 9 }
…略
```

第 7 行是增加的内容，置 allObjects 为 NULL。

header_obj.c 是对象头的实现，代码如下。

stepByStep/c3/a/object/header_obj.c

```
 1 #include "header_obj.h"
 2 #include "class.h"
 3 #include "vm.h"
 4
```




```
 5 DEFINE_BUFFER_METHOD(Value)
 6
 7 //初始化对象头
 8 void initObjHeader(VM* vm, ObjHeader* objHeader, ObjType objType, Class* class) {
 9     objHeader->type = objType;
10     objHeader->isDark = false;
11     objHeader->class = class;     //设置meta类
12     objHeader->next = vm->allObjects;
13     vm->allObjects = objHeader;
14 }
```

第 5 行的 DEFINE_BUFFER_METHOD(Value)生成与 ValueBuffer 配套的 4 个操作方法。

第 8 行定义的 initObjHeader 函数用于初始化参数 objHeader 指向的对象头，objType 是对象类型，class 是类。

第 9 行和第 11 行就是将参数对号入座，第 10 行 isDark 默认为 false，最终其值由垃圾收集机制设置。

第 12～13 行是将初始化的 objHeader 插入到链表头。

3.3 实现 class 定义（本节源码 stepByStep/c3/a）

class 包含两个文件，一个是 class.h，另一个是 class.c。不过本节只把 class.h 完成，class.c 所依赖的对象还没介绍，待后面完成相关依赖后一起说。class 是定义在 class.h 中的，代码如下。

stepByStep/c3/a/object/class.h

```
 1 #ifndef _OBJECT_CLASS_H
 2 #define _OBJECT_CLASS_H
 3 #include "common.h"
 4 #include "utils.h"
 5 #include "header_obj.h"
 6 #include "obj_string.h"
 7 #include "obj_fn.h"
 8
 9 typedef enum {
10     MT_NONE,      //空方法类型，并不等同于undefined
11     MT_PRIMITIVE,   //在vm中用C实现的原生方法
12     MT_SCRIPT,    //脚本中定义的方法
13     MT_FN_CALL,   //有关函数对象的调用方法，用来实现函数重载
14 } MethodType;     //方法类型
15
16 #define VT_TO_VALUE(vt) \
17     ((Value){vt, {0}})
18
19 #define BOOL_TO_VALUE(boolean) (boolean ? VT_TO_VALUE(VT_TRUE) : VT_TO_VALUE(VT_FALSE))
20 #define VALUE_TO_BOOL(value) ((value).type == VT_TRUE ? true : false)
21
22 #define NUM_TO_VALUE(num) ((Value){VT_NUM, {num}})
23 #define VALUE_TO_NUM(value) value.num
24
25 #define OBJ_TO_VALUE(objPtr) ({ \
```

```
    Value value; \
    value.type = VT_OBJ; \
    value.objHeader = (ObjHeader*)(objPtr); \
    value; \
})

#define VALUE_TO_OBJ(value) (value.objHeader)
#define VALUE_TO_OBJSTR(value) ((ObjString*)VALUE_TO_OBJ(value))
#define VALUE_TO_OBJFN(value) ((ObjFn*)VALUE_TO_OBJ(value))
#define VALUE_TO_OBJCLOSURE(value) ((ObjClosure*)VALUE_TO_OBJ(value))
#define VALUE_TO_CLASS(value) ((Class*)VALUE_TO_OBJ(value))

#define VALUE_IS_UNDEFINED(value) ((value).type == VT_UNDEFINED)
#define VALUE_IS_NULL(value) ((value).type == VT_NULL)
#define VALUE_IS_TRUE(value) ((value).type == VT_TRUE)
#define VALUE_IS_FALSE(value) ((value).type == VT_FALSE)
#define VALUE_IS_NUM(value) ((value).type == VT_NUM)
#define VALUE_IS_OBJ(value) ((value).type == VT_OBJ)
#define VALUE_IS_CERTAIN_OBJ(value, objType) (VALUE_IS_OBJ(value) && VALUE_TO_
OBJ(value)->type == objType)
#define VALUE_IS_OBJSTR(value) (VALUE_IS_CERTAIN_OBJ(value, OT_STRING))
#define VALUE_IS_OBJINSTANCE(value) (VALUE_IS_CERTAIN_OBJ(value, OT_INSTANCE))
#define VALUE_IS_OBJCLOSURE(value) (VALUE_IS_CERTAIN_OBJ(value, OT_CLOSURE))
#define VALUE_IS_OBJRANGE(value) (VALUE_IS_CERTAIN_OBJ(value, OT_RANGE))
#define VALUE_IS_CLASS(value) (VALUE_IS_CERTAIN_OBJ(value, OT_CLASS))
#define VALUE_IS_0(value) (VALUE_IS_NUM(value) && (value).num == 0)

//原生方法指针
typedef bool (*Primitive)(VM* vm, Value* args);

typedef struct {
    MethodType type;  //union 中的值由 type 的值决定
    union {
        //指向脚本方法所关联的 C 实现
        Primitive primFn;

        //指向脚本代码编译后的 ObjClosure 或 ObjFn
        ObjClosure* obj;
    };
} Method;

DECLARE_BUFFER_TYPE(Method)

//类是对象的模板
struct class {
    ObjHeader objHeader;
    struct class* superClass; //父类
    uint32_t fieldNum;        //本类的字段数，包括基类的字段数
    MethodBuffer methods;     //本类的方法
    ObjString* name;          //类名
```

```
75 };  //对象类
76
77 typedef union {
78     uint64_t bits64;
79     uint32_t bits32[2];
80     double num;
81 } Bits64;
82
83 #define CAPACITY_GROW_FACTOR 4
84 #define MIN_CAPACITY 64
85 #endif
```

方法有很多种，第 9 行定义的 MethodType 是方法类型，MT_PRIMITIVE 指的是用 C 语言原生实现的方法，MT_SCRIPT 是在脚本中定义的方法，MT_FN_CALL 是专用于函数调用的方法，主要用于重载，如 fun1.call()、fun1.call(x)、fun1.call(x，x)。

第 16 行定义的宏 VT_TO_VALUE 用于把 ValueType 类型转换为 Value 结构，很简单，读者参照前面介绍的 ValueType 和 Value 就明白了。

第 19～49 行定义的宏在 Value 结构和其他结构之间进行转换，XX_TO_VALUE 把 XX 转换为 Value 结构，VALUE_TO_XX 把 Value 转换为 XX 结构，VALUE_IS_XX 判断 Value 是否为某种特定的结构。

第 53 行定义的 Primitive 是原生方法的指针，在我们系统中定义的原生方法太多，以后就用此指针指向不同的方法统一调用。

第 55～64 行定义的 Method 是方法结构，type 是指方法的类型，其下 union 结构中的 primFn 是原生方法指针，脚本方法有可能在“幕后”是用 C 方法实现的，因此要用 primFn 指向此 C 方法，type 为 MT_PRIMITIVE 时，union 中 primFn 有效。obj 是指向脚本代码编译后的 ObjClosure 或 ObjFn，type 为 MT_SCRIPT 时，union 中 obj 有效。

第 66 行的 DECLARE_BUFFER_TYPE(Method)声明 MethodBuffer 类型。

第 69 行定义的 struct class 就是我们所说的类，之前说过，在 common.h 中有代码 typedef struct class Class，因此以后都用 Class 来表示类结构。

类也有自己的类，类的类就是 meta class，meta class 中存储的是类的元信息，将来会细说。既然类也有所属的类，那么它也要有对象头，因此在第 70 行添加了 objHeader。第 71 行的 superClass 指向 class 的基类。第 72 行的 fieldNum 是类中域的数量，此数量包括从基类继承的域。第 73 行的 methods 用来存储类中所有的方法，第 74 行的 name 是类名，其类型是 ObjString*，这就是我们所说的字符串对象，后面会介绍。

第 77～81 行定义的 Bits64 用来存储 64 位数据。

第 83 行定义的宏 CAPACITY_GROW_FACTOR 值为 4，作为 map 和 list 扩容时的系数。

第 84 行定义的宏 MIN_CAPACITY 为 64，后面介绍的 map 对象扩容时也会用到此项。

3.4 实现字符串对象（本节源码 stepByStep/c3/a）

本语言中一切皆对象，字符串也是。字符串字面量会被自动处理为字符串对象，比如源码 123 在词法分析时会被封装为字符串对象类型 ObjString。说明一下，尽管传统的创建对象的方式是 xxx.new()，但业界很少用这种方式创建字符串对象，因为此方式反而给用户带来了麻烦，比如用 String.new("123")这种形式创建字符串 123。

字符串对象定义在 object/obj_string.h 中，代码如下。

stepByStep/c3/a/object/obj_string.h

```
#ifndef _OBJECT_STRING_H
#define _OBJECT_STRING_H
#include "header_obj.h"

typedef struct {
    ObjHeader objHeader;
    uint32_t hashCode;  //字符串的哈希值
    CharValue value;
} ObjString;

uint32_t hashString(char* str, uint32_t length);
void hashObjString(ObjString* objString);
ObjString* newObjString(VM* vm, const char* str, uint32_t length);
#endif
```

第 5 行定义的 ObjString 便是字符串对象，对象都有自己的对象头 objHeader。

第 7 行定义的 hashCode 是字符串的哈希值，value 是“值”，不同对象的值有不同的含义，对于字符串对象来说，值就是字符串。其类型 CharValue 之前介绍过，定义在 utils.h 中。

接下来的是 3 行函数声明，它们定义在 obj_string.c，代码如下。

stepByStep/c3/a/object/obj_string.c

```
#include "obj_string.h"
#include <string.h>
#include "vm.h"
#include "utils.h"
#include "common.h"
#include <stdlib.h>

//fnv-1a 算法
uint32_t hashString(char* str, uint32_t length) {
    uint32_t hashCode = 2166136261, idx = 0;
    while (idx < length) {
        hashCode ^= str[idx];
        hashCode *= 16777619;
        idx++;
    }
    return hashCode;
}

//为 string 计算哈希码并将值存储到 string->hash
void hashObjString(ObjString* objString) {
    objString->hashCode =
        hashString(objString->value.start, objString->value.length);
}

//以 str 字符串创建 ObjString 对象,允许空串""
ObjString* newObjString(VM* vm, const char* str, uint32_t length) {
```

```
27      //length 为 0 时 str 必为 NULL length 不为 0 时 str 不为 NULL
28      ASSERT(length == 0 || str != NULL, "str length don't match str!");
29
30      //+1 是为了结尾的'\0'
31      ObjString* objString = ALLOCATE_EXTRA(vm, ObjString, length + 1);
32
33      if (objString != NULL) {
34          initObjHeader(vm, &objString->objHeader, OT_STRING, vm->stringClass);
35          objString->value.length = length;
36
37          //支持空字符串: str 为 null,length 为 0
38          //如果非空则复制其内容
39          if (length > 0) {
40              memcpy(objString->value.start, str, length);
41          }
42          objString->value.start[length] = '\0';
43          hashObjString(objString);
44      } else {
45          MEM_ERROR("Allocating ObjString failed!");
46      }
47      return objString;
48  }
```

第 9 行定义的函数 hashString 用于计算以 str 为起始，长度为 length 的字符串的哈希值，采用的算法是 fnv-1a。fnv-1a 算法特点是哈希速度快，结果高度分散，因此碰撞率较低。

第 20 行定义的函数 hashObjString 用于计算字符串对象 objString 的哈希值，并将结果存入到 objString->hashCode，实现上还是调用了 hashString。

第 26 行定义的函数 newObjString 用字符串 str 创建字符串对象。第 31 行调用 ALLOCATE_EXTRA 为新建的字符串对象 objString 分配内存，这里预留了结尾的“\0”，如果内存申请成功就在第 34 行调用函数 initObjHeader 初始化对象头，第 35 行用 length 更新字符串的长度即 objString→value.length。字符串对象允许空字符串，空字符串即“”。如果字符串非空就在第 39 行复制字符串到 value 的起始地址即 objString->value.start。第 42 行添加字符串结尾字符“\0”，在第 43 行调用 hashObjString 为其计算哈希值。如果之前内存申请失败就在第 45 行调用 MEM_ERROR 宏报错。最后返回创建的 objString。

3.5 模块对象和实例对象（本节源码 stepByStep/c3/a）

正如你正在看的文字，除了文字本身外，它还有一些修饰信息，比如字体大小、颜色等，像这种属于数据的数据、信息的信息称为元信息。在我们的内部也有一些类似的元信息，它们也是对象，我称它们为元对象（非官方命名，为便于叙述，我自己起的名字）。包括模块对象、实例对象等，这些元对象仅供内部使用，因此用户看不到。元对象定义在 meta_obj.h 中，代码如下。

stepByStep/c3/a/object/meta_obj.h

```
1 #ifndef _OBJECT_METAOBJ_H
2 #define _OBJECT_METAOBJ_H
3 #include "obj_string.h"
```

```
4
5 typedef struct {
6     ObjHeader  objHeader;
7     SymbolTable moduleVarName;    //模块中的模块变量名
8     ValueBuffer moduleVarValue;   //模块中的模块变量值
9     ObjString*  name;    //模块名
10 } ObjModule;    //模块对象
11
12 typedef struct {
13     ObjHeader objHeader;
14     //具体的字段
15     Value fields[0];
16 } ObjInstance;  //对象实例
17
18 ObjModule* newObjModule(VM* vm, const char* modName);
19 ObjInstance* newObjInstance(VM* vm, Class* class);
20 #endif
```

第 5 行定义的 ObjModule 是模块对象，模块就是在某独立作用域中定义的代码的集合。我们常说的导入“模块”就是指此模块的意义。说明一下，尽管模块是代码集合，但我们并不在其中保存代码（确切地说是代码对应的指令流），而是保存在代码单元中，代码单元就是后面介绍的函数对象，在函数对象中有专门存储指令的 instrStream，代码编译后存储到 instrStream 中。在模块对象中只保存其中的模块变量——在模块中定义的全局变量。第 7 行定义的 moduleVarName 用于存储本模块中所有模块变量的名字，第 8 行定义的 moduleVarValue 用于存储本模块中所有模块变量的值。

对象是类的实例，因此对象也称对象实例或实例对象。不同类的对象之间的不同，本质上体现在类定义的不同，而类是通用数据的模板，所有公用的部分即域和方法都放在类中了，对象调用方法时直接去它所属的类中找方法。介绍了 struc class 之后，你应该能猜到是去 class.Methods 中查找方法。而同一个类的不同对象之间的区别，仅体现在私有数据——域（即成员属性）。注意，在 sparrow 中，域和方法都没有访问权限，比如 C++中的 private、protect 和 public 我们并不支持。域只有静态域和实例域两种，方法也只有静态方法和实例方法两种，静态的域或方法属于类，被所有对象共享。对象自己的数据仅仅是域的值，这就是同类对象之间的区别所在，因此对于不同的对象只要单独维护各自域的值就行。对所有类（无论类定义相同或不同）的对象来说，它们的方法已经放在类中，那么剩下的区别是域的个数不同，数组正好满足此类需求，因此用一种结构就能表示对象实例了，这就是下行介绍的 ObjInstance。

第 12 行定义的 ObjInstance 是实例对象，是通用的对象结构，它包含的对象头用于查找它所属的类，第 15 行的 fields 是对象自己的域的值。提醒一下，域的数量属于通用数据，放在对象所属的类中了。重复一下，class 结构是存储类的方法的结构，ObjInstance 是存储对象的域的结构。此时你可能有疑问，对象是如何知道自己属于哪个类的？这是在前面介绍过的 initObjHeader 函数中完成的关联，调用 initObjHeader 时要指定参数 class，此 class 会被赋给对象头中的 class，具体代码为 objHeader->class = class，如果感到生疏，请重新看一下 header_obj.c 文件。

最后的是两个函数声明定义在 meta_obj.c，代码如下。

stepByStep/c3/a/object/meta_obj.c

```
1 #include "obj_fn.h"
2 #include "class.h"
```

```
 3 #include "vm.h"
 4 #include <string.h>
 5
 6 //新建模块
 7 ObjModule* newObjModule(VM* vm, const char* modName) {
 8     ObjModule* objModule = ALLOCATE(vm, ObjModule);
 9     if (objModule == NULL) {
10         MEM_ERROR("allocate ObjModule failed!");
11     }
12
13     //ObjModule 是元信息对象,不属于任何一个类
14     initObjHeader(vm, &objModule->objHeader, OT_MODULE, NULL);
15
16     StringBufferInit(&objModule->moduleVarName);
17     ValueBufferInit(&objModule->moduleVarValue);
18
19     objModule->name = NULL;    //核心模块名为 NULL
20     if (modName != NULL) {
21         objModule->name = newObjString(vm, modName, strlen(modName));
22     }
23
24     return objModule;
25 }
26
27 //创建类 class 的实例
28 ObjInstance* newObjInstance(VM* vm, Class* class) {
29     //参数 class 主要作用是提供类中 field 的数目
30     ObjInstance* objInstance = ALLOCATE_EXTRA(vm,
31           ObjInstance, sizeof(Value) * class->fieldNum);
32
33     //在此关联对象的类为参数 class
34     initObjHeader(vm, &objInstance->objHeader, OT_INSTANCE, class);
35
36     //初始化 field 为 NULL
37     uint32_t idx = 0;
38     while (idx < class->fieldNum) {
39        objInstance->fields[idx++] = VT_TO_VALUE(VT_NULL);
40     }
41     return objInstance;
42 }
```

第 7 行定义的函数 newObjModule 用于创建新模块，模块名是参数 modName，第 8 行为模块分配内存，第 14 行调用 initObjHeader 为其初始化对象头，注意，这里为第 4 个形参 class 传递的实参是 NULL，也就是使 objModule-> objHeader –>class 等于 NULL，这是因为 ObjModule 不属于任何一个类。第 16～17 行分别初始化 moduleVarName 和 moduleVarValue。ObjModule 中的成员 name 表示模块名，类型是字符串对象指针 ObjString*。我们规定，核心模块的模块名为 NULL，目的是避免与用户自定义模块的模块名冲突。第 19 行默认为核心模块，如果参数 modName 不为 NULL，说明不是核心模块，那么在第 21 行用 modName 作为参数调用 newObjString 创建字符串对象为其创建模块名。完成后在第 24 行返回创建的模块 objModule。

第 28 行定义的 newObjInstance 用于创建参数 class 的实例对象。实例对象 ObjInstance 中的 fields 数组虽然定义的大小为 0，但其起到的作用是柔性数组，创建 ObjInstance 时需要确定数组 fields 大小以确定需要分配的内存大小，数组大小是由参数 class->fieldNum 决定的（通用数据要放在模板即 class 中）。

第 34 行调用 initObjHeader 为其初始化对象头，第 4 个实参是 class，说明所创建的对象所属的类是此 class，也就是在此将对象和类做了绑定。第 37～40 行初始化数组 fields 中元素的值为 NULL。第 41 行返回 objInstance。

3.6 upvalue、openUpvalue 和 closedUpvalue

函数中是可以再定义函数的，因此函数之间就形成了包含的嵌套关系。包含其他函数定义的函数可称为外层函数，相应地被包含的函数就是内层函数。

upvalue 原本是 lua 闭包中的概念，upvalue 字面意思为“上值”，是指位于“上”层函数中的局部变量的“值”，这里的上层函数也就是外层函数。其实 upvalue 就是我们在第 0 章中介绍的闭包中所提到的自由变量。upvalue 是内部函数所引用的位于外层函数（即直接外层函数或间接外层函数）中的局部变量，比如下面伪码中的变量 outerVar1 和 outerVar2 都是函数 inner 的 upvalue。

```
funtion outer2() {
    var outerVar2;
    funtion outer1() {
        var outerVar1;
        funtion inner() {
            outerVar2++;
            outerVar1--;
            print outerVar1 + outerVar2
        }
    }
}
```

什么是 open upvalue 和 closed upvalue？既然这两个词都是修饰 upvalue 的，也就是其本质上还是外层函数中的局部变量，只不过该外层函数中的局部变量处于两种不同状态。

我们已经知道，局部变量是存在于运行时栈中的。如果内层函数所引用的局部变量仍然存储在该局部变量所属外层函数的运行时栈中，即该局部变量仍然可通过运行时栈来访问，那么对于内层函数来说，该局部变量被称为 open upvalue，这个 open 类似“景区开放中，可随时参观”的意思。如果函数执行完毕，也就是过了生存周期，那么函数的运行时栈也会被回收，位于栈中的局部变量当然也会一同消失，因此该局部变量变得不可访问。虽然该局部变量消失了，但毕竟还有内部函数引用它，必须妥善处理这种消失的局部变量，这就是闭包的作用。为了让内部函数可以正常访问到本已消失的局部变量，闭包机制会将该变量从栈中复制到一个安全的、且只有内部函数可以访问到的地方（在我们的实现中，这个安全的地方是 ObjUpvalue 结构中的 closed Upvalue 变量，下一节会介绍），这个已经消失的局部变量对于内部函数来说被称为 closed upvalue，类似“景区已关闭，不可参观”的意思。

本来到这儿就结束了，但有的读者比较严谨，说 closed upvalue 的解释有矛盾：“局部变量是存储在栈中的，如果内部函数引用的外层局部变量已不在外层函数的栈中，那还能叫局部变量吗？”这位读者说的很对，确实不能再称为局部变量了，这也正是把内层函数所引用的外层函数

的局部变量统称为 upvalue 的原因，可以把 upvalue 理解为指针，upvalue 指向的外层函数局部变量若在外层函数的栈中，该 upvalue 就被称为 open upvalue；若不在栈中，该 upvalue 就被称为 closed upvalue。

3.7 实现函数对象、闭包对象与调用框架（本节源码 stepByStep/c3/a）

尽管我们提供关键字 fun 以支持传统方式的函数定义，但函数在语言内部也是对象，对象都是通过“类名.new()”来产生的，以后你会知道我们把函数类的名字定义为 Fn（习惯于复合数据结构首字符大写），换句话说，可以通过 Fn.new()创建一个函数。

强调，对象是对象，类是类，不要搞混了，在本书及本系统中，与对象相关的标识符一律会显式带上 obj。本节我们要实现的是函数对象，即 ObjFn，这里所说的函数对象并不专指一般意义的函数，函数本身是一个独立的代码单元，代码最终也会被翻译为指令，因此函数最终呈现的是独立的指令集合单元，在此把指令集合称为“指令流”，我们的函数对象 ObjFn 正是代表指令流单元。在本系统中指令流单元有很多，比如模块是最大的指令流单元，类中的每个方法也都是独立的指令流单元，代码块也是指令流单元，闭包也是指令流单元，只要是指令流单元我们就用 ObjFn 来表示，因此 ObjFn 泛指一切指令流单元，函数也是指令流单元，将来我们也会用它来实现函数。为陈述方便，以后提到函数时，默认为指令流单元。

继续之前，需要在 vm 中新增 fnClass，代码如下。

stepByStep/c3/a/vm/vm.h

```
...略
 7 struct vm {
 8     Class* stringClass;
 9     Class* fnClass;
10     uint32_t allocatedBytes;  //累计已分配的内存量
11     ObjHeader* allObjects;  //所有已分配对象链表
12     Parser* curParser;  //当前词法分析器
13 };
...略
```

新增了第 9 行的 fnClass，后面的 newObjFn 函数会用到 fnClass。

下面介绍函数对象 ObjFn，它定义在 object/obj_fn.h 中，代码如下。

stepByStep/c3/a/object/obj_fn.h

```
 1 #ifndef _OBJECT_FN_H
 2 #define _OBJECT_FN_H
 3 #include "utils.h"
 4 #include "meta_obj.h"
 5
 6 typedef struct {
 7     char* fnName;      //函数名
 8     IntBuffer lineNo; //行号
 9 } FnDebug;    //在函数中的调试结构
10
11 typedef struct {
12     ObjHeader objHeader;
```

仅供非商业用途或交流学习使用


```
    ByteBuffer instrStream;  //函数编译后的指令流
    ValueBuffer constants;   // 函数中的常量表

    ObjModule* module;       //本函数所属的模块

    //本函数最多需要的栈空间,是栈使用空间的峰值
    uint32_t maxStackSlotUsedNum;
    uint32_t upvalueNum;   //本函数所涵盖的 upvalue 数量
    uint8_t argNum;   //函数期望的参数个数
#if DEBUG
    FnDebug* debug;
#endif
} ObjFn;   //函数对象

typedef struct upvalue {
    ObjHeader objHeader;

    //栈是个 Value 类型的数组,localVarPtr 指向 upvalue 所关联的局部变量
    Value* localVarPtr;

    //已被关闭的 upvalue
    Value closedUpvalue;

    struct upvalue* next;   //用以链接 openUpvalue 链表
} ObjUpvalue;   //upvalue 对象

typedef struct {
    ObjHeader objHeader;
    ObjFn* fn;   //闭包中所要引用的函数

    ObjUpvalue* upvalues[0]; //用于存储此函数的 closed upvalue
} ObjClosure;     //闭包对象

typedef struct {
    uint8_t* ip;     //程序计数器 指向下一个将被执行的指令

    //在本 frame 中执行的闭包函数
    ObjClosure* closure;

    //frame 是共享 therad.stack
    //此项用于指向本 frame 所在 thread 运行时栈的起始地址
    Value* stackStart;
} Frame;  //调用框架

#define INITIAL_FRAME_NUM 4

ObjUpvalue* newObjUpvalue(VM* vm, Value* localVarPtr);
ObjClosure* newObjClosure(VM* vm, ObjFn* objFn);
ObjFn* newObjFn(VM* vm, ObjModule* objModule, uint32_t maxStackSlotUsedNum);
#endif
```

仅供非商业用途或交流学习使用

开发一门语言最重要的部分是调试，第 6 行定义的 FnDebug 是调试结构，用于辅助指令流单元的调试。第 7 行的 fnName 是函数名，lineNo 是行号，我们在语义分析中生成指令时会用上此结构。

第 11～25 行定义的是函数对象 ObjFn，第 13 行的 instrStream 用于存储函数编译后的指令流。第 14 行的 constants 用于存储“函数”中的常量，称为常量表。你可能在想，函数中也会有常量？类似 C 语言中用 static 关键字修饰的静态变量？完全不是这样，我们语言中的函数是没有常量的，只有局部变量。前面说过，所有指令流单元都用 ObjFn 表示，因此 ObjFn 未必专指函数，ObjFn 是“通用的”指令流单元结构，函数虽然也是 ObjFn，但函数中没有常量，所以用不到 constants，此处的 constants 必然是为那些包含常量的指令流单元提供的，比如模块就是最大的指令流单元，位于模块中的模块变量、类的类名都会作为常量存储到模块中的常量表。第 16 行的 module 表示函数所属的模块。因为函数（包括类中的方法）有自己的形参和局部变量，所以这些数据要单独存储以形成独立的作用域，避免与其他函数中的数据冲突，传统且唯一的做法是把这些数据放在栈中，因为这些都是在函数执行阶段存在的，所以此栈被称为运行时栈。显而易见的是，我们在函数执行前要为其分配好运行时栈，否则函数无法运行，栈“通常”是内存中的结构（说“通常”的原因是，栈只是个概念，实现的方式不限，甚至可以用文件系统来实现，但毕竟太慢了，因此没人那么做），因此涉及分配的内存大小，那么栈需要多大的空间呢？第 19 行的 maxStackSlotUsedNum 是本函数最多需要的栈空间，是栈使用空间的峰值，这是在编译时统计的。第 20 行的 upvalueNum 是本函数所引用的 upvalue 的数量，upvalue 是位于本函数直接或间接外层函数中的局部变量，也就是自由变量的数量。第 21 行的 argNum 表示函数的形参个数。第 23 行的 debug 用于调试，只有定义了宏 DEBUG，成员 debug 才会被添加到 ObjFn 中。前面说过，宏 DEBUG 是在 makefile 中通过给 gcc 程序添加参数-DDEBUG 定义的，参数-D 用来定义一个宏，后面紧跟的 DEBUG 就是此宏。

第 27～37 行定义的 ObjUpvalue 是 upvalue 对象。第 31 行的变量 localVarPtr 用于指向所关联的 upvalue。upvalue 是直接外层或间接外层函数中的局部变量，局部变量肯定是位于栈中，因此通常情况下用变量 localVarPtr 指向栈中的 upvalue 以供引用该 upvalue 的内层函数使用。我们的栈是由 Value 数组实现的，因此 localVarPtr 的类型是 Value*。但那只是通常的情况，如果外层函数生存周期过了，那个外层函数的运行时栈也必然会被释放，upvalue 也就不存在了，在这种情况下，为了继续使用该 upvalue 会做两件事。

（1）upvalue 会从栈中被复制到第 34 行的 closedUpvalue 中。

（2）使第 31 行的指针变量 localVarPtr 指向 closedUpvalue 的地址。

经过以上两步骤，使得引用该 upvalue 的函数依然可以通过 localVarPtr 访问到正确的结果，即原来外层函数中的局部变量。

第 36 行的 next 是 upvalue 指针，用以形成 openUpvalue 链表。

第 39～44 行定义的 ObjClosure 是闭包对象，闭包就是“函数”及其依赖的“环境”，函数是指第 41 行的函数对象 fn，环境是指第 43 行的 upvalues 数组。

第 46～55 行定义的是函数的调用帧栈，也称调用框架，帧栈是函数的一套运行环境，包括：

（1）程序计算器 PC。也就是下一条指令的地址，在 x86 体系中指的是 cs:ip 寄存器，因此第 47 行我们也用 ip 表示下一条指令地址。

（2）待运行的闭包。一般情况下在帧栈中运行的都是函数，可函数如果引用了 upvalue 就变成了闭包，因此第 50 行的 closure 就是指待运行的闭包。

（3）要运行函数必然要提供运行时栈，因此“帧栈”中的“栈”便是此意义，第 54 行的 stackStart 就是函数运行时栈的起始地址。

仅供非商业用途或交流学习使用

补充说明，为了在一个线程中运行多个函数，线程中有一个大的运行时栈，该运行时栈被线程中所有函数共享，每个函数自己的运行时栈都是起始于线程这个大运行时栈的某个地址的一片连续空间。

第 57 行的 INITIAL_FRAME_NUM 是线程中初始化的 Frame 数量，目前为 4 个。

后面的 3 个函数声明定义在 obj_fn.c 中，obj_fn.c 代码如下。

stepByStep/c3/a/object/obj_fn.c

```
#include "meta_obj.h"
#include "class.h"
#include "vm.h"

//创建一个空函数
ObjFn* newObjFn(VM* vm, ObjModule* objModule, uint32_t slotNum) {
    ObjFn* objFn = ALLOCATE(vm, ObjFn);
    if (objFn == NULL) {
        MEM_ERROR("allocate ObjFn failed!");
    }
    initObjHeader(vm, &objFn->objHeader, OT_FUNCTION, vm->fnClass);
    ByteBufferInit(&objFn->instrStream);
    ValueBufferInit(&objFn->constants);
    objFn->module = objModule;
    objFn->maxStackSlotUsedNum = slotNum;
    objFn->upvalueNum = objFn->argNum = 0;
#ifdef DEBUG
    objFn->debug = ALLOCATE(vm, FnDebug);
    objFn->debug->fnName = NULL;
    IntBufferInit(&objFn->debug->lineNo);
#endif
    return objFn;
}

//以函数 fn 创建一个闭包
ObjClosure* newObjClosure(VM* vm, ObjFn* objFn) {
    ObjClosure* objClosure = ALLOCATE_EXTRA(vm,
          ObjClosure, sizeof(ObjUpvalue*) * objFn->upvalueNum);
    initObjHeader(vm, &objClosure->objHeader, OT_CLOSURE, vm->fnClass);
    objClosure->fn = objFn;

    //清除 upvalue 数组，以避免在填充 upvalue 数组之前触发 GC
    uint32_t idx = 0;
    while (idx < objFn->upvalueNum) {
        objClosure->upvalues[idx] = NULL;
        idx++;
    }

    return objClosure;
}

//创建 upvalue 对象
```

仅供非商业用途或交流学习使用


```
43 ObjUpvalue* newObjUpvalue(VM* vm, Value* localVarPtr) {
44     ObjUpvalue* objUpvalue = ALLOCATE(vm, ObjUpvalue);
45     initObjHeader(vm, &objUpvalue->objHeader, OT_UPVALUE, NULL);
46     objUpvalue->localVarPtr = localVarPtr;
47     objUpvalue->closedUpvalue = VT_TO_VALUE(VT_NULL);
48     objUpvalue->next = NULL;
49     return objUpvalue;
50 }
```

第 6 行定义的函数 newObjFn 用于创建函数对象，形参 objModule 表示函数所属的模块，slot 意为槽位，任何数组结构中的一个数据单位都可称为 slot，比如一个数组元素就是一个 slot，形参 slotNum 表示函数需要的栈的大小。

第 26 行定义的函数 newObjClosure，用参数 objFn 创建一个闭包对象。前面比较简单，第 29 行用 initObjHeader 初始化对象头时，实参是 vm->fnClass，这个是 vm 中新增的类的挂载点。说一下第 33～37 行，将来我们会实现垃圾回收，垃圾回收简称 GC，即 Garbage Collection，GC 在运行时会标记闭包中 upvalues 数组的每个元素，如果元素为 NULL，则相应的处理函数直接返回，并不处理。因此把 upvalues 数组元素逐个显式置为 NULL，否则未初始化元素的值是未知的，GC 会误处理为 upvalue 从而造成严重错误。

第 43 行定义的 newObjUpvalue 函数用于创建 upvalue 对象。形参 localVarPtr 是局部变量在栈中的地址，将其在第 46 行赋予 objUpvalue->localVarPtr。第 47 行将 objUpvalue->closedUpvalue 用宏 VT_TO_VALUE 初始化为 VT_NULL 的 Value 形式。将 next 置为 NULL 后返回创建的 objUpvalue。

目前只是完成了有关对象的基础部分，有了这些基础才能继续完善词法分析器。尽管本节代码（stepByStep/c3/a）原本可以编译成功，但我们并未把新功能添加到词法分析器中，parser 依然是上节（stepByStep/c2/f）中的功能，运行时没有变化容易给人造成困扰，因此我把 makefile 和 parser/token.list 删除了。token.list 只是目前为演示词法分析临时加的，将来也要删掉。本节代码由于删除了 token.list，编译时会报 tokenArray 未声明，请读者知晓。

3.8 完善词法分析器之数字解析（本节源码 stepByStep/c3/b）

数字和字符串的处理尚未完成，本节先搞定数字部分。继续介绍之前先贴出 parser.h 中变化的内容，最新的 Token 结构和 Parser 结构如下所列。

stepByStep/c3/b/parser/parser.h

```
...略
85 typedef struct {
86     TokenType type;
87     const char* start;
88     uint32_t length;
89     uint32_t lineNo;
90     Value value;
91 } Token;
92
93 struct parser {
94     const char* file;
95     const char* sourceCode;
```

仅供非商业用途或交流学习使用


```
    const char* nextCharPtr;
    char curChar;
    Token curToken;
    Token preToken;
    ObjModule* curModule;     //当前正在编译的模块

    //处于内嵌表达式之中时,期望的右括号数量
    //用于跟踪小括号对儿的嵌套
    int interpolationExpectRightParenNum;
    VM* vm;
};
…略
```

第 90 行在 Token 结构中新增了 value 成员，用于存储单词的值，比如对于数字字符串我们会用 value 来存储其数值，对于字符串我们也会用它来存储字符串对象的 Value 结构，后面会看到。

第 100 行在 Paser 结构中新增了 curModule 成员，用于记录当前正在编译的模块，后面很多情况需要知道当前正在编译哪个模块。

下面是完善 parser 使其支持数字解析，包括八进制、十进制、十六进制数字，还有之前未彻底完成的字符串解析。这里只列出了有变化的代码，其他相同的部分一律用“…略”跳过，若不想翻看源码文件的话可以到前面小节中找相同的部分。

stepByStep/c3/b/parser/parser.c

```
…略
#include "obj_string.h"
…略
//解析十六进制数字
static void parseHexNum(Parser* parser) {
    while (isxdigit(parser->curChar)) {
        getNextChar(parser);
    }
}

//解析十进制数字
static void parseDecNum(Parser* parser) {
    while (isdigit(parser->curChar)) {
        getNextChar(parser);
    }

    //若有小数点
    if (parser->curChar == '.' && isdigit(lookAheadChar(parser))) {
        getNextChar(parser);
        while (isdigit(parser->curChar)) { //解析小数点之后的数字
            getNextChar(parser);
        }
    }
}

//解析八进制
static void parseOctNum(Parser* parser) {
```

仅供非商业用途或交流学习使用


```
122     while(parser->curChar >= '0' && parser->curChar < '8') {
123         getNextChar(parser);
124     }
125 }
126
127 //解析八进制、十进制、十六进制，仅支持前缀形式,后缀形式不支持
128 static void parseNum(Parser* parser) {
129     //十六进制的 0x 前缀
130     if (parser->curChar == '0' && matchNextChar(parser, 'x')) {
131         getNextChar(parser);  //跳过 x
132         parseHexNum(parser);    //解析十六进制数字
133         parser->curToken.value =
134             NUM_TO_VALUE(strtol(parser->curToken.start, NULL, 16));
135     } else if (parser->curChar == '0'
136             && isdigit(lookAheadChar(parser))) {  // 八进制
137         parseOctNum(parser);
138         parser->curToken.value =
139             NUM_TO_VALUE(strtol(parser->curToken.start, NULL, 8));
140     } else {        //解析十进制
141         parseDecNum(parser);
142         parser->curToken.value = NUM_TO_VALUE(strtod(parser->curToken.start, NULL));
143     }
144     //nextCharPtr 会指向第一个不合法字符的下一个字符,因此-1
145     parser->curToken.length =
146         (uint32_t)(parser->nextCharPtr - parser->curToken.start - 1);
147     parser->curToken.type = TOKEN_NUM;
148 }
…略
```

以下几种进制的数字解析是由后面介绍的数字解析入口函数 parseNum 调用的，因此在各种进制数字解析函数的实现中并未判断进制前缀，由 parseNum 统一处理。

第 99 行定义的函数 parseHexNum 用于解析十六进制数字，循环调用 isxdigit 函数，判断字符若为十六进制就连续读入字符。

第 106 行定义的 parseDecNum 用于解析十进制数字。第 107～109 行循环调用 isdigit 函数，判断若为整数就连续读取，直到遇到非数字字符。我们支持十进制浮点数，因此在第 112 行判断上面循环中遇到的非数字字符是否为小数点，如果是小数点并且小数点后面的字符是数字字符（用 lookAheadChar 返回下一个字符），那么说明是浮点数，在第 113～116 行继续循环读取数字字符。

第 121 行定义的 parseOctNum 函数用于解析八进制数字。八进制数字范围是 0～7，因此在第 122 行做了范围判断。

第 128 行定义的 parseNum 函数是数字解析的入口，根据不同的进制前缀调用上面的 3 种进制数字解析函数。第 130 行判断，如果是前缀 0x，说明是十六进制数字，由于调用 matchNextChar 后，curChar 变成 x，因此在第 130 行调用 getNextChar 跳过字符 x，剩余的字符才是数字，第 132 行调用 parseHexNum 处理其余十六进制数字。完成之后，在第 133 行调用 NUM_TO_VALUE 把数字转换为 Value，存储到 curToken.value。简要说下函数 strtol，它是将第一个参数指定的字符串按照第 3 个参数指定的进制转换为长整型。只有数字字符才能被转换为数字，非数字字符在转换为数字时肯定失败，第二个参数 NULL 表示不需要返回转换失败的字符串地址。第 135～139 行

仅供非商业用途或交流学习使用

判断是否是八进制数字，八进制数字以 0 开头，方法相同。第 141～142 行用于处理十进制数字，函数 strtod 是将参数 1 指定的字符串转换为十进制数字，NULL 表示不返回转换失败的字符地址。第 145 行计算数字单词的长度，第 147 行将类型置为 TOKEN_NUM。

3.9 完善词法分析器之字符串解析和获取 token（本节源码 stepByStep/c3/b）

距离最终的词法分析器还有一小步，即字符串解析的收尾工作以及在 getNextToken 函数中更新对数字的识别。代码如下。

stepByStep/c3/b/parser/parser.c

```
…略
184 //解析字符串
185 static void parseString(Parser* parser) {
186     ByteBuffer str;
187     ByteBufferInit(&str);
…略
254     //用识别到的字符串新建字符串对象存储到 curToken 的 value 中
255     ObjString* objString = newObjString(parser->vm, (const char*)str.datas, str.count);
256     parser->curToken.value = OBJ_TO_VALUE(objString);
257     ByteBufferClear(parser->vm, &str);
258 }
…略
298 //获得下一个 token
299 void getNextToken(Parser* parser) {
300     parser->preToken = parser->curToken;
301     skipBlanks(parser);  // 跳过待识别单词之前的空格
302     parser->curToken.type = TOKEN_EOF;
303     parser->curToken.length = 0;
304     parser->curToken.start = parser->nextCharPtr - 1;
305     parser->curToken.value = VT_TO_VALUE(VT_UNDEFINED);
…略
430             default:
431                 //处理变量名及数字
432                 //进入此分支的字符肯定是数字或变量名的首字符
433                 //后面会调用相应函数把其余字符一并解析
434
435                 //首字符是字母或_则是变量名
436                 if (isalpha(parser->curChar) || parser->curChar == '_') {
437                     parseId(parser, TOKEN_UNKNOWN);  //解析变量名其余的部分
438                 } else if (isdigit(parser->curChar)) { //数字
439                     parseNum(parser);
440                 } else {
441                     if (parser->curChar == '#' && matchNextChar(parser, '!')) {
442                         skipAline(parser);
443                         parser->curToken.start = parser->nextCharPtr - 1;
                            //重置下一个 token 起始地址
444                         continue;
```

仅供非商业用途或交流学习使用


```
445                 }
446                 LEX_ERROR(parser, "unsupport char: \'%c\', quit.", parser->curChar);
447             }
448             return;
449         }
450         //大部分 case 的出口
451         parser->curToken.length = (uint32_t)(parser->nextCharPtr - parser->curToken.start);
452         getNextChar(parser);
453         return;
454     }
455 }
...略
483 //由于 sourceCode 未必来自于文件 file,有可能只是个字符串,
484 //file 仅用作跟踪待编译的代码的标识,方便报错
485 void initParser(VM* vm, Parser* parser, const char* file,
486                 const char* sourceCode, ObjModule* objModule) {
487     parser->file = file;
488     parser->sourceCode = sourceCode;
489     parser->curChar = *parser->sourceCode;
490     parser->nextCharPtr = parser->sourceCode + 1;
491     parser->curToken.lineNo = 1;
492     parser->curToken.type = TOKEN_UNKNOWN;
493     parser->curToken.start = NULL;
494     parser->curToken.length = 0;
495     parser->preToken = parser->curToken;
496     parser->interpolationExpectRightParenNum = 0;
497     parser->vm = vm;
498     parser->curModule = objModule;
499 }
```

之前我们还没有实现 value 及其依赖的结构，如今已经在 token 中添加了 value 成员，因此本次很多更新都是与 token 中的 value 相关。

首先更新的是第 185 行的函数 parseString，在其开头定义的 str 用来收集识别到的字符串，在上一个版本的 parseString 中并没有将收集到 str 中的字符串利用上，本次新增的第 255～256 行的代码便是以 str 中的字符串新建字符串对象，并将该字符串对象通过 OBJ_TO_VALUE 转换为 Value 结构存储到 curToken.value 中。

下面更新的是第 299 行的函数 getNextToken，在第 305 行通过宏 VT_TO_VALUE 把 curToken.value 初始化为 VT_UNDEFINED 的 Value 结构形式，此 token 中的 value 只会被数字和字符串使用，识别到数字或字符串后 vaule 自然会被更正为正确的类型，因此这里的显式初始化并不是必需的。

（1）确保识别到数字和字符串后，token.value 的类型不再是 VT_UNDEFINED，而是 VT_OBJ 或 VT_NUM。

（2）确保在识别数字和字符串之外的字符串时不被之前识别到的数字和字符串的结果误导。

第 438～439 行判断首字符若为数字的话就调用 parseNum 解析数字。十六进制和八进制数字的前缀都是字符 0，因此也会识别到。

最后一个要更新的是第 485 行的函数 initParser，其声明中添加了形参 objModule，表示正在解析的模块。然后在第 498 行用 objModule 初始化 parser 中的 curModule。

由于 initParser 的声明变了，调用 initParser 的地方也要修改，在 cli/cli.c 中为了临时添加了

NULL 作为形参 objModule 的实参，代码如下。

stepByStep/c3/b/cli/cli.c

```
…略
 8 //执行脚本文件
 9 static void runFile(const char* path) {
21     struct parser parser;
22     initParser(vm, &parser, path, sourceCode, NULL);  //此 NULL 是临时的
23
24     #include "token.list"
…略
```

目前的词法分析器已经是最终版了，下节是功能验证。

3.10 最终版词法分析器的功能验证（本节源码 stepByStep/c3/b）

本节是功能验证，下面是样本文件。

stepByStep/c3/b/sample.sp

```
 1 import people for People
 2 fun fn() {
 3     var p = People.new("xiaoming", "male", 20.0)
 4     p.sayHi()
 5 }
 6
 7 class Family < People {
 8     var father
 9     var mother
10     var child
11     new(f, m, c) {
12         father = f
13         mother = m
14         child  = c
15         super("wbf", "male", 60)
16     }
17 }
18
19 var f = Family.new("wbf", "ls", "shine")
20 f.sayHi()
21
22 fn()
```

其实还是之前的样本文件，只是第 3 行和第 15 行分别加了浮点数和整数，在 c3/b 目录下执行 make，编译之后运行，下面是其输出。

```
[work@work b]$ ./spr sample.sp
1L: IMPORT [import]
1L: ID [people]
1L: FOR [for]
```

```
1L: ID [People]
2L: FUN [fun]
2L: ID [fn]
2L: LEFT_PAREN [(]
2L: RIGHT_PAREN [)]
2L: LEFT_BRACE [{]
3L: VAR [var]
3L: ID [p]
3L: ASSIGN [=]
3L: ID [People]
3L: DOT [.]
3L: ID [new]
3L: LEFT_PAREN [(]
3L: STRING ["xiaoming"]
3L: COMMA [,]
3L: STRING ["male"]
3L: COMMA [,]
3L: NUM [20.0]
3L: RIGHT_PAREN [)]
4L: ID [p]
4L: DOT [.]
4L: ID [sayHi]
4L: LEFT_PAREN [(]
4L: RIGHT_PAREN [)]
5L: RIGHT_BRACE [}]
7L: CLASS [class]
7L: ID [Family]
7L: LESS [<]
7L: ID [People]
7L: LEFT_BRACE [{]
8L: VAR [var]
8L: ID [father]
9L: VAR [var]
9L: ID [mother]
10L: VAR [var]
10L: ID [child]
11L: ID [new]
11L: LEFT_PAREN [(]
11L: ID [f]
11L: COMMA [,]
11L: ID [m]
11L: COMMA [,]
11L: ID [c]
11L: RIGHT_PAREN [)]
11L: LEFT_BRACE [{]
12L: ID [father]
12L: ASSIGN [=]
12L: ID [f]
13L: ID [mother]
13L: ASSIGN [=]
```

```
13L: ID [m]
14L: ID [child]
14L: ASSIGN [=]
14L: ID [c]
15L: SUPER [super]
15L: LEFT_PAREN [(]
15L: STRING ["wbf"]
15L: COMMA [,]
15L: STRING ["male"]
15L: COMMA [,]
15L: NUM [60]
15L: RIGHT_PAREN [)]
16L: RIGHT_BRACE [}]
17L: RIGHT_BRACE [}]
19L: VAR [var]
19L: ID [f]
19L: ASSIGN [=]
19L: ID [Family]
19L: DOT [.]
19L: ID [new]
19L: LEFT_PAREN [(]
19L: STRING ["wbf"]
19L: COMMA [,]
19L: STRING ["ls"]
19L: COMMA [,]
19L: STRING ["shine"]
19L: RIGHT_PAREN [)]
20L: ID [f]
20L: DOT [.]
20L: ID [sayHi]
20L: LEFT_PAREN [(]
20L: RIGHT_PAREN [)]
22L: ID [fn]
22L: LEFT_PAREN [(]
22L: RIGHT_PAREN [)]
23L: EOF []
[work@work b]$
```

词法分析器到此结束。

3.11 实现 list 列表对象（本节源码 stepByStep/c3/c）

list 即列表，我们的列表同 Python 中的 list 列表是一回事。在我们语言中列表定义有两种形式，一种是字面量形式，如 var li = ["a", "b", "c"]，注意我们不支持字符，只支持字符串，因此列表中各元素要用双引号括起来。另一种建立列表的形式是对象模式，如“var li = List.new()”，访问元素如 li[1]表示获取 list 中索引为 1 的元素，即"b"。我们同样支持列表分片，分片的意思是从列表中“分”出一“片”，因此分片的结果还是列表，只不过是小于等于原列表的子集，切记，分片的结果集中即使只包括一个元素，那也是列表，绝不是单个字符。Python 中的分片运算符是冒号:，在：左边的是分片的起始索引 from，在：右边的是分片终止索引 to，如 Python 代码 li[0:2]。我们

的分片运算符是下节介绍的..，它表示 range 对象。先简单说一下 range 对象，range 对象是由其属性 from 和 to 描述的一段步进为 1 的范围，同样，from 和 to 分别位于..的左右两边，如 sparrow 代码 li[0..2]。不过与 Python 不同的是，我们的 range 对象的使用场合不仅是在下标运算符即中括号[]中，由于它表示的范围是一个可迭代的序列，因此还可以用在需要序列的环境中，比如 for 循环的迭代序列中：for i (1..3)，变量 i 的值会分别为 1、2、3。你也看到了，range 对象所表示的范围包括 to，因此我们的 li[0..2]表示列表["a","b","c"]。如果你对 Python 了解，你就会知道其 li[0:2]表示['a','b']。

list 定义在 object/obj_list.h 中，代码如下。

stepByStep/c3/c/object/obj_list.h

```
 1 #ifndef _OBJECT_LIST_H
 2 #define _OBJECT_LIST_H
 3 #include "class.h"
 4 #include "vm.h"
 5
 6 typedef struct {
 7     ObjHeader objHeader;
 8     ValueBuffer elements;  //list 中的元素
 9 } ObjList;  //list 对象
10
11 ObjList* newObjList(VM* vm, uint32_t elementNum);
12 Value removeElement(VM* vm, ObjList* objList, uint32_t index);
13 void insertElement(VM* vm, ObjList* objList, uint32_t index, Value value);
14 #endif
```

第 6～9 行定义的 ObjList 是列表对象，其中的成员 elements 用于存储列表中所有元素。elements 类型是 ValueBuffer，列表元素也是值，“值”的统一形式就是 Value。

list 有关的增、删、改操作在 obj_list.c 文件中，如下所示。

stepByStep/c3/c/object/obj_list.c

```
 1 #include "obj_list.h"
 2
 3 //新建 list 对象,元素个数为 elementNum
 4 ObjList* newObjList(VM* vm, uint32_t elementNum) {
 5     //存储 list 元素的缓冲区
 6     Value* elementArray = NULL;
 7
 8     //先分配内存,后调用 initObjHeader,避免 gc 无谓地遍历
 9     if (elementNum > 0) {
10         elementArray = ALLOCATE_ARRAY(vm, Value, elementNum);
11     }
12     ObjList* objList = ALLOCATE(vm, ObjList);
13
14     objList->elements.datas = elementArray;
15     objList->elements.capacity = objList->elements.count = elementNum;
16     initObjHeader(vm, &objList->objHeader, OT_LIST, vm->listClass);
17     return objList;
18 }
19
20 //在 objlist 中索引为 index 处插入 value, 类似于 list[index] = value
```

```
21 void insertElement(VM* vm, ObjList* objList, uint32_t index, Value value) {
22     if (index > objList->elements.count - 1) {
23         RUN_ERROR("index out bounded!");
24     }
25
26     //准备一个 Value 的空间以容纳新元素产生的空间波动
27     //即最后一个元素要后移 1 个空间
28     ValueBufferAdd(vm, &objList->elements, VT_TO_VALUE(VT_NULL));
29
30     //下面使 index 后面的元素整体后移一位
31     uint32_t idx = objList->elements.count - 1;
32     while (idx > index) {
33        objList->elements.datas[idx] = objList->elements.datas[idx - 1];
34        idx--;
35     }
36
37     //在 index 处插入数值
38     objList->elements.datas[index] = value;
39 }
40
41 //调整 list 容量
42 static void shrinkList(VM* vm, ObjList* objList, uint32_t newCapacity) {
43     uint32_t oldSize = objList->elements.capacity * sizeof(Value);
44     uint32_t newSize = newCapacity * sizeof(Value);
45     memManager(vm, objList->elements.datas, oldSize, newSize);
46     objList->elements.capacity = newCapacity;
47 }
48
49 //删除 list 中索引为 index 处的元素,即删除 list[index]
50 Value removeElement(VM* vm, ObjList* objList, uint32_t index) {
51     Value valueRemoved = objList->elements.datas[index];
52
53     //使 index 后面的元素前移一位,覆盖 index 处的元素
54     uint32_t idx = index;
55     while (idx < objList->elements.count) {
56        objList->elements.datas[idx] = objList->elements.datas[idx + 1];
57        idx++;
58     }
59
60     //若容量利用率过低就减小容量
61     uint32_t _capacity = objList->elements.capacity / CAPACITY_GROW_FACTOR;
62     if (_capacity > objList->elements.count) {
63         shrinkList(vm, objList, _capacity);
64     }
65
66     objList->elements.count--;
67     return valueRemoved;
68 }
```

第 4 行定义的 newObjList 函数用于新建元素个数为 elementNum 的列表对象。

允许列表为空，即 elementNum 为 0，如果列表不为空，则在第 10 行为元素缓冲区 elementArray 分配内存，elementArray 用于存储列表中的元素。这里先为其分配内存是有好处的，这与垃圾回收有关。垃圾回收简称 GC，GC 是在内存分配时判断总的内存使用量到达阈值时触发，将来就是在内存分配函数 memManager 中添加判断的代码。GC 处理（或回收）的是通过 initObjHeader 函数挂载到 vm->allObjects 链表中的对象，要对各个对象中的各个结构全部遍历一次，以后你会知道，GC 的清扫阶段会通过此链表查找白色对象，整个遍历过程的工作量还是蛮大的。此时先分配内存，后调用 initObjHeader 的好处是，若 gc 也是在此时触发，此时尚未将新建的 list 对象挂载到 vm->allObjects，因此 GC 在清扫阶段少处理了一个对象，相当于提高了 GC 的效率。不要小看一个对象，对象中的数据若比较复杂，比如其组成又包括其他对象，对 GC 影响是很大的，GC 执行时往往会 stop the world，这个 world 是指整个解析器，即垃圾回收启动时，整个程序不得不先暂停，因此当然希望执行垃圾回收的时间越短越好。第 12～17 行是一些内存分配及初始化工作，提醒一下，各种定义的 XXXBuffer 中用于存储数据的是其中的 datas 指向的缓冲区，因此第 14 行代码将 elementArray 赋值给 objList->elements.datas。

第 21 行定义的 insertElement 函数用于在 objlist 中的索引 index 处插入 value，就是脚本中代码 objlist[index]=value 的内部实现。

insertElement 只支持一个元素的插入，在新元素插入前，index 后面的元素都要往后移一位，第 28 行先往 objList->elements 添加一个元素，为插入新元素引起的空间后移准备空间。

第 31～35 行的循环使 index 后面的元素整体后移一位，腾出空间后在第 38 行完成插入。

第 42 行定义的函数 shrinkList 用于调整 objList 的容量为 newCapacity，容量即最大可容纳的元素数量。调整容量到容量对应的内存大小，也就是调用 memManager 为列表元素缓冲区 objList->elements.datas 调整内存大小。

第 50 行定义的 removeElement 函数用于删除 objList 中索引为 index 的元素，即删除 objList[index]。

元素删除后，其位置不能空着，逻辑需求上要把 index 后面的元素整体左移一个位置，即代码第 54～58 行完成的工作。

宏 CAPACITY_GROW_FACTOR 的值为 4，第 61～64 行判断如果容量利用率过低，即列表中实际元素使用的空间不足列表容量的 1/4，就调用上面的 shrinkList 调整列表空间。

第 66 行 objList->elements.count--;即减少一个元素。

列表对象到此结束。

3.12 range 对象（本节源码 stepByStep/c3/c）

range 对象表示一段步进为 1 数字范围，它是用..隐式创建的，并没有显式的 new 形式。range 对象是个闭区间，如 1..8 表示 1,2,3,4,5,6,7,8。range 既然是个范围，肯定有起始和终止元素，首元素就是起始元素，简称 from，尾元素是终止元素，简称 to。

range 用于索引列表等对象，from 和 to 的值最终会被转换成列表中的元素下标，比如 to 若为 -1 则表示尾元素，from 未必要小于 to，当 from 大于 to 时表示逆序，如['a', 'b', 'c', 'd'] [3..1]表示['d', 'c', 'b']，当 from 等于 to 时，['a', 'b', 'c', 'd'] [2..2]表示['c']。

range 定义在头文件 obj_range.h 中，代码如下。

stepByStep/c3/c/object/obj_range.h

```
1 #ifndef _OBJECT_RANGE_H
2 #define _OBJECT_RANGE_H
```

```
 3 #include "class.h"
 4
 5 typedef struct {
 6     ObjHeader objHeader;
 7     int from;    //范围的起始
 8     int to;      //范围的结束
 9 } ObjRange;      //range 对象
10
11 ObjRange* newObjRange(VM* vm, int from, int to);
12 #endif
```

第 5～9 行定义的 ObjRange 就是 range 对象，from 和 to 分别是 range 的首尾元素。

下面是其实现文件 obj_range.c，代码如下。

stepByStep/c3/c/object/obj_range.c

```
 1 #include "obj_range.h"
 2 #include "utils.h"
 3 #include "class.h"
 4 #include "vm.h"
 5
 6 //新建 range 对象
 7 ObjRange* newObjRange(VM* vm, int from, int to) {
 8     ObjRange* objRange = ALLOCATE(vm, ObjRange);
 9     initObjHeader(vm, &objRange->objHeader, OT_RANGE, vm->rangeClass);
10     objRange->from = from;
11     objRange->to = to;
12     return objRange;
13 }
```

本文件中只有一个函数 newObjRange，用于创建从 from 到 to 的 range 对象，分配内存后初始化其对象头，用参数 from 和 to 初始化对象的 from 和 to，最后返回。

3.13 迟到的 class.c（本节源码 stepByStep/c3/c）

以往说明各种文件时，通常都是.h 和.c 文件一起介绍，唯独介绍 class.h 时没说 class.c，因为当时涉及一些对象没有介绍，现在可以看 class.c 的实现了，现在也只是初步实现，以后功能将逐步完善。

查看 class.c 之前先看下最新的 class.h，代码如下。

stepByStep/c3/c/object/class.h

```
...略
32 #define VALUE_TO_OBJ(value) (value.objHeader)
33 #define VALUE_TO_OBJSTR(value) ((ObjString*)VALUE_TO_OBJ(value))
34 #define VALUE_TO_OBJFN(value) ((ObjFn*)VALUE_TO_OBJ(value))
35 #define VALUE_TO_OBJRANGE(value) ((ObjRange*)VALUE_TO_OBJ(value))
36 #define VALUE_TO_OBJINSTANCE(value) ((ObjInstance*)VALUE_TO_OBJ(value))
37 #define VALUE_TO_OBJLIST(value) ((ObjList*)VALUE_TO_OBJ(value))
38 #define VALUE_TO_OBJMAP(value) ((ObjMap*)VALUE_TO_OBJ(value))
39 #define VALUE_TO_OBJCLOSURE(value) ((ObjClosure*)VALUE_TO_OBJ(value))
40 #define VALUE_TO_OBJTHREAD(value) ((ObjThread*)VALUE_TO_OBJ(value))
```

```
41 #define VALUE_TO_OBJMODULE(value) ((ObjModule*)VALUE_TO_OBJ(value))
42 #define VALUE_TO_CLASS(value) ((Class*)VALUE_TO_OBJ(value))
...略
```

新增了一些对象，在第 35～41 行增加了新对象的宏。就是从 Value 到不同对象之间的转换。下面是 class.c，代码如下。

stepByStep/c3/c/object/class.c

```
1  #include "class.h"
2  #include "common.h"
3  #include "string.h"
4  #include "obj_range.h"
5  #include "core.h"
6  #include "vm.h"
7
8  DEFINE_BUFFER_METHOD(Method)
9
10 //判断 a 和 b 是否相等
11 bool valueIsEqual(Value a, Value b) {
12     //类型不同则无需进行后面的比较
13     if (a.type != b.type) {
14         return false;
15     }
16
17     //同为数字,比较数值
18     if (a.type == VT_NUM) {
19         return a.num == b.num;
20     }
21
22     //同为对象,若所指的对象是同一个则返回 true
23     if (a.objHeader == b.objHeader) {
24         return true;
25     }
26
27     //对象类型不同无需比较
28     if (a.objHeader->type != b.objHeader->type) {
29         return false;
30     }
31
32     //以下处理类型相同的对象
33     //若对象同为字符串
34     if (a.objHeader->type == OT_STRING) {
35         ObjString* strA = VALUE_TO_OBJSTR(a);
36         ObjString* strB = VALUE_TO_OBJSTR(b);
37         return (strA->value.length == strB->value.length &&
38                 memcmp(strA->value.start, strB->value.start, strA->value.length) == 0);
39     }
40
41     //若对象同为 range
42     if (a.objHeader->type == OT_RANGE) {
43         ObjRange* rgA = VALUE_TO_OBJRANGE(a);
```

```
44          ObjRange* rgB = VALUE_TO_OBJRANGE(b);
45          return (rgA->from == rgB->from && rgA->to == rgB->to);
46      }
47
48      return false;  //其他对象不可比较
49  }
```

第 8 行的宏 DEFINE_BUFFER_METHOD(Method)用于定义 Method 缓冲区 MethodBuffer 相应的操作函数。

第 11 行定义的 valueIsEqual 函数用于判断 a 和 b 两个 Value 是否相等。

比较的前提是类型一致的情况下再比较值。在第 13 行判断类型如果不同就返回 false。第 18 行若判断为数字，就比较数值。第 23 行判断 a 和 b 都是对象类型且对象头 objHeader 所指的是同一个对象则返回 true。第 28 行判断若对象类型不同就返回 false。之后便是处理类型相同的对象，第 34 行判断若是字符串对象，先将 a 和 b 转换为字符串对象指针 strA 和 strB，判断这两个字符串对象的字符串长度及字符串是否全相等，此处判断长度的目的是避免其中一个字符串是另一个字符串的前缀，如“abc”是“abcd”的前缀，那么仅通过 memcmp 判断前 3 个字符就误判了。第 42～46 行判断 range 对象，原理是比较 a 和 b 两个对象的 from 和 to 是否相等。

class.c 暂时就这样。

3.14 map 对象（本节源码 stepByStep/c3/c）

map 对象被称为关联数组、哈希数组、字典等多个名字。

3.14.1　哈希表

哈希表属于数据结构中的内容，读者都学过数据结构，因此我就不再搬出数据结构中的那一套了，说下自己的认识。

当人处于认知积累阶段的时候就是初学者，初学者肯定习惯接触具体实在的东西以帮助他们建立对世界的认识。数据结构是抽象的，因此很多读者初次接触数据结构时，完全不知所云，甚至由于数据结构的教材往往都以 C 语言为实现，很多读者误以为数据结构就是 C 语言的高级应用。无论是否懂数据结构，编程中肯定离不开它。与数据相关的就是存储，数据就是从一个地方到另一个地方被存储来存储去，既然称为“数据”结构，那么自然离不开数据的存储形式，比如是用变量还是数组，这已经是数据结构的领域了，数据结构中的“结构”在此就是指变量或数组。尽管不懂，但却一直在用它，感受不到它的存在是因为缺少对它的认识。

数据结构是组织数据的方法，这包括数据存储的方式及配套的数据获取方式。不同的存储方式其获取数据的方式自然也不同。

本节要介绍的是哈希表。结构是由需求决定的，什么样的需求要用到哈希表呢？就是快速地查找。哈希表的时间复杂度是 O(1)，已经没法再快了，因此它是实现快速查找的最佳方式。哈希表是一张数据表，前面说过了，数据结构包括数据的存储和配套的获取方式。哈希表负责存储，用来存储关键字的数据，表中容纳数据的单位空间称为槽位，即 slot，槽位只是个概念，并不局限于某种事物。数据结构中存储数据的单位都可称为槽位，它来自于硬件中的插槽 slot。对于表来说，槽位就是表中的一行。对哈希表进行“实例化”就容易理解了，表可以用线性结构来实现，因此可以用数组来实现哈希表，数组元素就是槽位。与哈希表配套的操作方法是哈希方法，哈希方法是一个函数，输入参数是关键字和哈希表，返回值是该关键字在哈希表中的地址，即哈希地

址，因此哈希函数是计算关键字在哈希表中地址的方法。具体来说，哈希地址是哈希表中槽位的索引。

哈希算法的初衷是把关键字离散到单独的槽位中，理想情况是一个槽位只被一个关键字占据，比如哈希表中有 10 个槽位，即哈希表大小是 10，如果有 10 个数，哈希函数希望把这 10 个数分别放在 10 个槽位中。当关键字数小于哈希表大小时，只要哈希足够均匀肯定没问题，但关键字数量大于哈希表大小时肯定就违背初衷了，也就是说，即使哈希算法保证了离散均匀分布，查找效率也还会受限于哈希表的容量，当关键字数量大于哈希表容量时必然会出现同一个槽位被再次分配给其他关键字的情况，这就是同义词冲突。

同义词是指具有相同哈希值的关键字，在哈希表中，哈希值就是指哈希地址，也就是关键字的哈希地址冲突了，因此两个关键字落到了同一个槽位中。哈希方法有很多，常见的有拉链法和开放定址法，这两种方法的不同之处在于处理同义词的方式，哈希算法是固定的，哈希表容量是有限的，因此同义词是无法避免的，好的哈希算法是通过算法本身让冲突率最低。拉链法是在该槽位中建立一个链表，把所有同义词用这个链表连接起来，查找时再从链表中遍历所有节点逐一比较关键字。而拉链法中的一个槽位容纳了很多关键字，为了更加形象表示“多”，这种容纳多个关键字的槽位便称为“桶”，即哈希桶，哈希桶是拉链法解决关键字冲突的容器。刚说过，理想情况下哈希表的一个槽位只容纳一个关键字，开放定址法就是这样的，但它是“半理想”，由于同义词是无法避免的，开放定址法采取了妥协退让的方式，开放定址法也称开放探测法，“探测”指的是当出现同义词时会调用另外一个探测函数寻找下一个可用的槽位。有关哈希表的详细内容可参考数据结构，我们也采取开放探测法。

3.14.2 map 对象头文件及 entry

读者都知道，很多语言都提供键（key）和值（value）相关联的数组，也称关联数组，它和普通数组的区别是，该数组是用 key 作为数组元素的索引，比如数组 age 用来存储姓名和年纪，age["xiaoming"]=20 表示小明年纪是 20 岁。因为关联数组在内部大多数通过哈希函数将 key 与 value 关联在一起的，因此亦称为哈希数组。当然实现这种关联数组的方法有很多，未必要采取哈希算法，但毕竟哈希算法的时间复杂度已经是 O(1)了。哈希函数建立的是一种映射关系，现实中具有映射关系的事物往往在语言中作为哈希数组的名字，比如字典 dict 表示字与其意义的映射关系，因此 dict 用在 Python 中表示哈希数组。地图 map 是地图上地标与实际地理的映射关系，因此 map 用在 C++中表示哈希数组，我们这里也用关键字 map 来表示哈希数组，它就是本节要实现的 map 对象。

map 对象定义在 object/obj_map.h 中，代码如下。

stepByStep/c3/c/object/obj_map.h

```
#ifndef _OBJECT_MAP_H
#define _OBJECT_MAP_H
#include "header_obj.h"

#define MAP_LOAD_PERCENT 0.8

typedef struct {
    Value key;
    Value value;
} Entry;    //key->value 对儿

```

```
12 typedef struct {
13     ObjHeader objHeader;
14     uint32_t capacity; //Entry 的容量(即总数)，包括已使用和未使用 Entry 的数量
15     uint32_t count;  //map 中使用的 Entry 的数量
16     Entry* entries; //Entry 数组
17 } ObjMap;
18
19 ObjMap* newObjMap(VM* vm);
20
21 void mapSet(VM* vm, ObjMap* objMap, Value key, Value value);
22 Value mapGet(ObjMap* objMap, Value key);
23 void clearMap(VM* vm, ObjMap* objMap);
24 Value removeKey(VM* vm, ObjMap* objMap, Value key);
25 #endif
```

第 8 行定义的宏 MAP_LOAD_PERCENT 表示 map 装载率，其实就是 map 的容量利用率。容量利用率是指容器（比如水杯）中已使用的空间与容器的最大空间之比再乘以 100%，此概念适合任何具有容器性质的事物，抽象来说就是使用量与总量之比，在此 MAP_LOAD_ PERCENT 值为 0.8 表示 80%，扩容时会以它为参考。

第 7～10 行定义的 Entry 是 key->value（即键->值）对儿，即一对键值就是一个 entry，entry 是条目、项的意思，哈希数组中的数组元素就是 entry。entry 可以简单地理解为容器中可列举的数据单位，比如文件系统中的目录项就叫 entry。

第 12～17 行定义的 ObjMap 是 map 对象，entries 是哈希数组中的 entry 数组。

3.14.3　冲突探测链与伪删除

我们采用的哈希方法是开放探测法，这涉及冲突探测链和伪删除，继续介绍 map 对象之前先把这两个概念交待清楚。

什么是冲突探测链呢？我们已经了解，在开放探测法中添加一个 key 时，如果 key 对应的 slotA 被其他同义词占用，会探测下一个 slotB，如果 slotB 还是被占用，继续寻找，直到找到可用的 slotC 为止（当然，这是在哈希表未写满的情况下）。我们把 slotA->slotB->slotC 称为冲突探测链。它们在哈希表中的情形如图 3-2 所示。

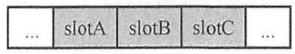


图 3-2

slot 中存储的是 entry，为方便陈述，姑且认为 entry 就是 slot。以上是写 key 的过程，下次读取该 key 时，必须使用同样的哈希算法才能找到该 key 所在的 slot，也就是说，在读取过程中依然会遍历一次冲突探测链：从 slotA 到 slotB，最后到 slotC（读者先不要考虑遍历冲突探测链的细节）。冲突探测链虽然不是链表，但相当于链表，因为它也像链表一样有明确的尾节点——由于冲突探测链中的 slot 肯定都是使用中的 slot，因此沿着冲突链直到遇到空闲的 slot，就说明冲突探测链结束了，这个空闲的 slot 便相当于此链表的尾节点。什么样的 slot 才算是空闲的 slot？总该有个识别的标记吧？是的，在我们的实现中，如果存储在 slot 中 entry 的 key 的 type 为 VT_UNDEFINED，那么这个 slot 即是空闲的。

下面是检索 key 的一般流程。

倘若 key 就在哈希表中，也存在该 key 的同义词，通过遍历冲突探测链上所有同义词的 slot，

逐个比对其中的 key，一定能找到该 key 正确的 slot，找到时便是遍历结束时。

假如 key 不在哈希表中，之前又插入了该 key 的同义词，就是说也产生了冲突探测链，我们读取冲突探测链上的 slot 肯定找不到该 key，而系统又不知道这个链有多长，因此哈希算法会继续调用探测函数算出下一个 slot（即 entry），我们经过比对，发现此 slot 是空闲的，至此遍历结束，说明找不到该 key，系统提示找不到该 key。

以上这两种情况使哈希表一切正常。对于哈希表来说，有插入就有删除，如果删除了某个 key，而该 key 又对应于冲突探测链上的 slotB，key 被删除了，它相应的 slot 自然也要被清空，也就是把该 slot 置为空闲的状态，结果如图 3-3 所示。

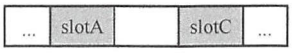


图 3-3

假设 slotC 的 key 是 keyC，那么在查找 keyC 时，由于 slotB 的删除导致了冲突探测链的断裂，在沿着冲突探测链遍历到原来的 slotB 时发现是空闲的，则说明探测链结束了，这导致找不到 keyC，系统会报错：keyC 不存在。其实 keyC 不是不存在，而是丢了。

问题的根源在于冲突探测链上被删除的 slot 擦除得太干净了，必须得留下点“标记”让别人知道这并不是冲突探测链的结束，后面还有同义词的 slot 呢。切记以下两点。

（1）我们是在辨别空闲 slot 时出的错，因此标记是留在空闲 slot 中的。

（2）该空闲 slot 空闲的原因是被删除过，是曾经被使用过，并不是从未被使用过的、绝对干净的 slot。

为解决这个问题，引入了“伪删除”，也就是在空闲的 slot 中留下标记，我们约定（为了让读者注意到标记的“重点”是空闲 slot）：

（1）slot 中 entry 的 key 的 type 若为 VT_UNDEFINED，则表示是空闲 slot，这是大前提，如果此时 entry 的 value 的 type 为 VT_TRUE，则表示探测链未断，可继续探测；

（2）slot 中 entry 的 key 的 type 若为 VT_UNDEFINED，则表示是空闲 slot，这是大前提，如果此时 entry 的 value 的 type 为 VT_FALSE，则表示探测链结束，探测结束。

也就是说，伪删除的标记就是空闲 slot 中 entry 的 value 为 VT_TRUE。那么在以后删除某 key 时，将其 slot 中的 entry.value.type 置为 VT_TRUE 就行了。那什么情况下 entry.value.type 的值为 VT_FALSE 呢？是那些未使用过的 slot，因为我们会把 map 中所有 entry 的 value 的 type 初始化为 VT_FALSE。那些使用过的 slot 被清空时，一律将 entry.value.type 置为 VT_TRUE。

梳理一下，时刻要清楚，删除的 slot 中 entry.key.type 肯定是 VT_UNDEFINED，以上两项规则的约定是为了避免已删除的 slot 造成冲突探测链断裂，其应用场合是在“读 slot”时的状态，即查找 key 时，如果对应的 slot 中 entry 的 key 的 type 为 VT_UNDEFINED，则首先说明此 slot 是空闲的，空闲的 slot 有两种，一种是未使用过的，另一种是曾经使用过后来被删除回收的。对于后者，在读的过程中一定要考虑此 slot 是否处于冲突探测链中，因此要额外观察 entry 的 value 的 type，如果其值为 VT_TRUE，说明此 slot 是冲突探测链中已经被删除的 slot，应该继续顺着冲突探测链向下找，如果其值为 VT_FALSE，说明此 slot 是全新的，因此绝不可能处于冲突探测链中。而对于“写”的状态，即添加 key 或修改 key 的值时，只需要关注 entry 的 key 的 type 为 VT_UNDEFINED 就行了，因为不用担心冲突探测链断裂，不管待写入的 slot 是否处于某个冲突探测链中，只要用实际的 value 去覆盖该 entry 中的 value 即可，毕竟已经删除的 slot 就应该被重新使用，因此不需要考虑原来 value 的 type 是否为 VT_TRUE，覆盖该 slot 后，该 entry 的 key 的 type 肯定就不再是 VT_UNDEFINED，因此不会为“读”key 时造成冲突探测链的困扰。此 slot

被覆盖后即处于使用中的状态，以后在“读”key 时可用来比对是否为待查找 key 的 entry。

3.14.4　map 对象的实现

下面是 map 对象的实现，定义在 obj_map.c 中，代码如下。

stepByStep/c3/c/object/obj_map.c (1/4)

```
#include "obj_map.h"
#include "class.h"
#include "vm.h"
#include "obj_string.h"
#include "obj_range.h"

//创建新 map 对象
ObjMap* newObjMap(VM* vm) {
    ObjMap* objMap = ALLOCATE(vm, ObjMap);
    initObjHeader(vm, &objMap->objHeader, OT_MAP, vm->mapClass);
    objMap->capacity = objMap->count = 0;
    objMap->entries = NULL;
    return objMap;
}

//计算数字的哈希码
static uint32_t hashNum(double num) {
    Bits64 bits64;
    bits64.num = num;
    return bits64.bits32[0] ^ bits64.bits32[1];
}

//计算对象的哈希码
static uint32_t hashObj(ObjHeader* objHeader) {
    switch (objHeader->type) {
       case OT_CLASS:  //计算 class 的哈希值
          return hashString(((Class*)objHeader)->name->value.start,
                 ((Class*)objHeader)->name->value.length);

       case OT_RANGE: { //计算 range 对象哈希码
          ObjRange* objRange = (ObjRange*)objHeader;
          return hashNum(objRange->from) ^ hashNum(objRange->to);
       }
       case OT_STRING:  //对于字符串,直接返回其 hashCode
          return ((ObjString*)objHeader)->hashCode;
       default:
          RUN_ERROR("the hashable are objstring, objrange and class.");
    }
    return 0;
}

//根据 value 的类型调用相应的哈希函数
static uint32_t hashValue(Value value) {
    switch (value.type) {
```

```
45          case VT_FALSE:
46              return 0;
47          case VT_NULL:
48              return 1;
49          case VT_NUM:
50              return hashNum(value.num);
51          case VT_TRUE:
52              return 2;
53          case VT_OBJ:
54              return hashObj(value.objHeader);
55          default:
56              RUN_ERROR("unsupport type hashed!");
57      }
58      return 0;
59  }
```

第 8 行定义的函数 newObjMap 用于创建新的 map 对象，前三行是分配内存，初始化对象头，这里的 vm->mapClass 是在 vm.h 中新增的，后面会列出。紧接着 capacity 和 count 初始化为 0，存储 entry 的数组 entries 初始化为 NULL，未分配内存的原因是目前 map 为空，随意分配内存会浪费，应以后在 map 中添加数据时根据实际情况酌情分配，添加数据就是对 map 执行 set 操作，如代码 m["key"]=value，具体实现是后面介绍的 mapSet 函数。

第 17 行的 hashNum 函数用于计算数字的哈希值，此前定义的 64 位结构 Bits64 正是用在此处，它用形参 num 高 32 位和低 32 位的异或结果做 num 的哈希值。

第 24 行定义的函数 hashObj 用于计算对象的哈希值。此函数用于处理 Class、ObjRange 和 ObjString。第 26～28 行用 hashString 函数计算 ObjString*类型的 Class->name。第 30～33 行处理 ObjRange，用 ObjRange 的 from 和 to 的哈希值再做异或，结果作为 range 对象的返回值。第 34～35 行处理 ObjString，直接返回字符串对象的哈希值 hashCode。

第 43 行定义的 hashValue 函数用于计算 value 的哈希值，也就是针对不同类型的 Value 调用不同的方法或直接返回数值。

stepByStep/c3/c/object/obj_map.c (2/4)

```
60
61  //在 entries 中添加 entry,如果是新的 key 则返回 true
62  static bool addEntry(Entry* entries, uint32_t capacity, Value key, Value value) {
63      uint32_t index = hashValue(key) % capacity;
64
65      //通过开放探测法去找可用的 slot
66      while (true) {
67          //找到空闲的 slot,说明目前没有此 key,直接赋值返回
68          if (entries[index].key.type == VT_UNDEFINED) {
69              entries[index].key = key;
70              entries[index].value = value;
71              return true;        //新的 key 就返回 true
72          } else if (valueIsEqual(entries[index].key, key)) { //key 已经存在,仅仅更新值就行
73              entries[index].value = value;
74              return false;  // 未增加新的 key 就返回 false
75          }
76
```

```
77          //开放探测定址,尝试下一个 slot
78          index = (index + 1) % capacity;
79      }
80  }
81
82  //使对象 objMap 的容量调整到 newCapacity
83  static void resizeMap(VM* vm, ObjMap* objMap, uint32_t newCapacity) {
84      // 1. 先建立一个新的 entry 数组
85      Entry* newEntries = ALLOCATE_ARRAY(vm, Entry, newCapacity);
86      uint32_t idx = 0;
87      while (idx < newCapacity) {
88          newEntries[idx].key = VT_TO_VALUE(VT_UNDEFINED);
89          newEntries[idx].value = VT_TO_VALUE(VT_FALSE);
90          idx++;
91      }
92
93    // 2. 再遍历老的数组,把有值的部分插入到新数组
94      if (objMap->capacity > 0) {
95          Entry* entryArr = objMap->entries;
96          idx = 0;
97          while (idx < objMap->capacity) {
98              //该 slot 有值
99              if (entryArr[idx].key.type != VT_UNDEFINED) {
100                 addEntry(newEntries, newCapacity,
101                         entryArr[idx].key, entryArr[idx].value);
102             }
103             idx++;
104         }
105     }
106
107     // 3. 将老 entry 数组空间回收
108     DEALLOCATE_ARRAY(vm, objMap->entries, objMap->count);
109     objMap->entries = newEntries;      //更新指针为新的 entry 数组
110     objMap->capacity = newCapacity;    //更新容量
111 }
```

希望读者还记得我们的 Entry 结构，它包括类型为 Value 的两个成员 key 和 value，也就是说 Entry 不仅要存储值，还要存储键，初学者可能会觉得奇怪，不是通过 key 哈希（在此为动词）到 value 所在的 slot 就行了吗？为什么还要存储 key 呢？你说的那是没有同义词冲突的“理想”的情况，大多数哈希表的 entry 都包含“额外”的 key，这是哈希算法本身的问题导致的，即同义词冲突，因此可能需要哈希函数“退而求其次”地找寻其他 slot 作为 key“真正”的容身之所，比如拉链法和开放探测法等。总之不能保证每个 key 都落在“理想的”slot 中，或者说不保证一个 slot 中只能有一个 key，因此这么做的目的是同义词比较时判断是否找到了 key 所在的“真正”的 slot。

第 62 行定义的函数 addEntry 用于在 entries 中添加 entry，如果是新的 key 则返回 true。此函数是用于往 map 中 set 数据时调用。第 63 行用 key 的哈希值再对哈希表容量 capacity 取模得到槽位索引 index，后面用此 index 在 entries 数组中获取 entry 后再做下一步判断。注意，此 key 的哈希值 hashValue(key)并不是上面所说哈希算法的输出，因为它只是针对某个数，并不针对哈希表，hashValue 仅用来计算某个数的哈希值，只是此哈希表的哈希算法中的一部分，再加上后面第 78

行的探测方法才是完整的哈希算法。第 66～79 行通过开放探测法去找可用的 slot。entries 是 entry 的数组，entry 中包括 key 和 value，即键和值都存储在 entry 中。第 68 行判断若 entries[index]中 key 的类型为 VT_UNDEFINED，说明此 slot 尚未存储任何 key，是空闲的可以使用，第 69～70 行设置值后，第 71 行返回 true 表示这是新的 key。注意，这里不需要把 type 的值 VT_UNDEFINED 更新为其他值，因为 type 不属于 entry 而属于 key。key 是 Value 结构，已经被设置好了。后面你会知道，所有 entry 的 key 我们会初始化为 VT_UNDEFINED。

第 72 行判断若 key 和该槽位中的 key 相等，说明 key 已经存在，这是为已有的 key 更新 value，因此第 73 行直接把 value 赋给 entries[index].value，由于不是新建的 key，第 74 行返回 false。如果槽位 entries[index]已经被占用了，第 78 行使 index 加 1 后对容量 capacity 取模，探测下一个槽位是否可用。

第 83 行定义的 resizeMap 函数使对象 objMap 的容量调整到 newCapacity。容量调整的思路是把原数据复制到新的空间后再回收旧空间，也许你会问，entries 数组为什么不直接原地扩容呢？像列表函数 shrinkList 那样调用 memManager 原地调整不行吗？答案是肯定不行的，因为尽管哈希表可以用线性结构实现，但其存储数据的方式可不是线性的，数据所在的 slot 是利用线性结构的容量取模计算得到的，属于离散分布，因此在新的容量下相同的哈希算法也会使同一个 key 落向不同的 slot，为了在新容量下能找到原来 key 的 value，只能把所有 key 和 value 重新按哈希算法插入到新空间，也就是说新数据不能保证写入到新扩容的空间中，必须要把原来的旧数据先 get 出来，扩容后再重新 set 一遍，这才保证了原来的旧数据可以在扩容后的存储系统中找到，这正是生产环境中 key-value 型存储系统扩容很麻烦的原因。第 85 行先分配容量为 newCapacity 的新 entry 数组，用此新空间存储原来的 entry。第 86～91 行循环初始化所有 entry 中的 key 为 VT_UNDEFINED，value 为 VT_FALSE，这看上去是合理的，也是有意为之的，后面再说。第 94 行遍历旧的 entries 数组，调用 addEntry 把原值转移到新空间 newEntries。第 108 行调用 DEALLOCATE_ARRAY 回收旧的数组 entries，第 109～110 行更新 entries 和 capacity。

stepByStep/c3/c/object/obj_map.c (3/4)

```
112
113 //在 objMap 中查找 key 对应的 entry
114 static Entry* findEntry(ObjMap* objMap, Value key) {
115     //objMap 为空则返回 null
116     if (objMap->capacity == 0) {
117         return NULL;
118     }
119
120 //以下开放定址探测
121 //用哈希值对容量取模计算槽位(slot)
122     uint32_t index = hashValue(key) % objMap->capacity;
123     Entry* entry;
124     while (true) {
125         entry = &objMap->entries[index];
126
127         //若该 slot 中的 entry 正好是该 key 的 entry,找到返回
128         if (valueIsEqual(entry->key, key)) {
129             return entry;
130         }
131
132         //key 为 VT_UNDEFINED 且 value 为 VT_TRUE 表示探测链未断,可继续探测
133         //key 为 VT_UNDEFINED 且 value 为 VT_FALSE 表示探测链结束,探测结束
```

```
134         if (VALUE_IS_UNDEFINED(entry->key) && VALUE_IS_FALSE(entry->value)) {
135             return NULL;     //未找到
136         }
137
138         //继续向下探测
139         index = (index + 1) % objMap->capacity;
140     }
141 }
142
143 //在 objMap 中实现 key 与 value 的关联:objMap[key]=value
144 void mapSet(VM* vm, ObjMap* objMap, Value key, Value value) {
145     //当容量利用率达到 80%时扩容
146     if (objMap->count + 1 > objMap->capacity * MAP_LOAD_PERCENT) {
147         uint32_t newCapacity = objMap->capacity * CAPACITY_GROW_FACTOR;
148         if (newCapacity < MIN_CAPACITY) {
149             newCapacity = MIN_CAPACITY;
150         }
151         resizeMap(vm, objMap, newCapacity);
152     }
153
154     //若创建了新的 key 则使 objMap->count 加 1
155     if (addEntry(objMap->entries, objMap->capacity, key, value)) {
156         objMap->count++;
157     }
158 }
159
160 //从 map 中查找 key 对应的 value: map[key]
161 Value mapGet(ObjMap* objMap, Value key) {
162   Entry* entry = findEntry(objMap, key);
163   if (entry == NULL) {
164       return VT_TO_VALUE(VT_UNDEFINED);
165   }
166   return entry->value;
167 }
```

第 114 行定义的 findEntry 函数用于在 objMap 中查找 key 对应的 entry，若容量为空则返回 NULL。之前是用 addEntry 写 entry，现在是用 findEntry 读 entry，这说明 findEntry 必须和 addEntry 使用同样的哈希算法才能找到某个 key 对应的 entry 所在的 slot。第 121～140 行就是采用同样的开放探测法遍历冲突的 entry。第 122 行 index 是 key 的哈希地址，即 key 对应的 entry 在哈希表中的索引，第 128 行判断如果找到该 key 就返回该 entry。我们目前是在“读”的阶段，因此会涉及冲突探测链，第 134 行就是根据之前的两项约定做判断，如果是空闲 slot 并且 entry 的 value 的 type 为 VT_FALSE，则说明冲突探测链结束，没找到结果，因此返回 NULL。到了第 139 行，说明要么 entry->key.type 不为 VT_UNDEFINED，要么 entry->value.type 为 VT_TRUE，这两种情况都说明在冲突探测中，继续探测。

第 144 行定义的 mapSet 函数用于在 objMap 中实现 key 与 value 的关联，即 objMap[key]=value 的实现。新创建的 ObjMap 没有为其 entries 数组分配空间，前面说过了允许空 map 存在，随意预分配是浪费内存，真正的内存分配是在 set 时，也就是第 146 行判断当容量利用率达到 80%时就扩容，map 初次的内存分配也正是此时。宏 MIN_CAPACITY 值为 64（定义在 class.h），MAP_LOAD_

PERCENT 值为 0.8（定义在 obj_map.h），CAPACITY_GROW_FACTOR 值为 4（定义在 obj_map.h）。分配内存或扩容后，第 155 行调用 addEntry 添加 entry，如果 addEntry 返回为 true，则说明是添加了新 key，使 count++。

第 161 行定义的 mapGet 函数用于从 map 中查找 key 对应的 value，即 map[key]的实现。先是调用 findEntry 获得 key 的 entry，如果 entry 为 NULL 则返回未定义，否则返回 entry 的 value，即 key 对应的 value。

stepByStep/c3/c/object/obj_map.c (4/4)

```
168
169 //回收 objMap.entries 占用的空间
170 void clearMap(VM* vm, ObjMap* objMap) {
171     DEALLOCATE_ARRAY(vm, objMap->entries, objMap->count);
172     objMap->entries = NULL;
173     objMap->capacity = objMap->count = 0;
174 }
175
176 //删除 objMap 中的 key,返回 map[key]
177 Value removeKey(VM* vm, ObjMap* objMap, Value key) {
178     Entry* entry = findEntry(objMap, key);
179
180     if (entry == NULL) {
181         return VT_TO_VALUE(VT_NULL);
182     }
183
184 //设置开放定址的伪删除
185     Value value = entry->value;
186     entry->key = VT_TO_VALUE(VT_UNDEFINED);
187     entry->value = VT_TO_VALUE(VT_TRUE);    //值为真,伪删除
188
189     objMap->count--;
190     if (objMap->count == 0) { //若删除该 entry 后 map 为空就回收该空间
191         clearMap(vm, objMap);
192     } else if (objMap->count < objMap->capacity / (CAPACITY_GROW_FACTOR) * MAP_LOAD_
        PERCENT &&
193             objMap->count > MIN_CAPACITY) {    //若 map 容量利用率太低,就缩小 map 空间
194         uint32_t newCapacity = objMap->capacity / CAPACITY_GROW_FACTOR;
195         if (newCapacity < MIN_CAPACITY) {
196             newCapacity = MIN_CAPACITY;
197         }
198         resizeMap(vm, objMap, newCapacity);
199     }
200
201     return value;
202 }
```

第 170 行定义的 clearMap 用于回收 objMap.entries 占用的空间，先回收内存，再初始化。

第 177 行定义的 removeKey 用于删除 objMap 中的 key，返回其对应的 value。第 178 行获取 key 所在的 entry，第 185 行获得 key 的 value，为第 201 行的返回值做准备。第 186～187 行就是伪删除，将 VT_UNDEFINED 写入 entry->key.type，将 VT_TRUE 写入 entry->value.type。删除一

个 key 后第 189 行使 count 减 1。第 190～199 行就是调整 map 容量大小，第 190 行如果 map 空了就回收空间，第 192 之后就是缩减容量的判断。最后第 201 行返回 key 的 value。

map 对象说完了，vm.h 中有更新的内容，代码如下。

stepByStep/c3/c/vm/vm.h

```
…略
 9 typedef enum vmResult {
10     VM_RESULT_SUCCESS,
11     VM_RESULT_ERROR
12 } VMResult;    //虚拟机执行结果
13 //如果执行无误,可以将字符码输出到文件缓存,避免下次重新编译
14
15 struct vm {
16     Class* classOfClass;
17     Class* objectClass;
18     Class* stringClass;
19     Class* mapClass;
20     Class* rangeClass;
21     Class* listClass;
22     Class* nullClass;
23     Class* boolClass;
24     Class* numClass;
25     Class* fnClass;
26     Class* threadClass;
27     uint32_t allocatedBytes;   //累计已分配的内存量
28     ObjHeader* allObjects;     //所有已分配对象链表
29     SymbolTable allMethodNames;    //(所有)类的方法名
30     ObjMap* allModules;
31     ObjThread* curThread;      //当前正在执行的线程
32     Parser* curParser;         //当前词法分析器
33 };
…略
```

第 9～12 行定义的 VMResult 是虚拟机执行的结果，后面增加了一些内容读者可自己查看。

3.15 线程对象（本节源码 stepByStep/c3/c）

3.15.1　线程、协程浅述

为什么要有线程的概念呢？原因是起初操作系统运行在单 CPU 核心上，由于只有一个 CPU，为了实现“同时”运行多个任务，工程师采取的是所有任务轮换使用 CPU 的策略，因为是轮换着用，所以只是伪并行，这种伪并行就是并发。并行和并发都是单位量，并行是同时进行的任务数量，并发是一定时间内运行的任务数量，是个累计量，比如 QA 读者经常做压力测试，常用高并发请求工具对服务器进行压测，最后出测试报告称：“并发一万，服务器负载较低，页面响应正常”。这里所说的“并发一万”是指单位时间内的总共请求量，比如压测工具 ab 就会指定在多少秒内完成这些请求量，并不是真正同时发出的请求，即使是一秒的时间，对于速度极快的计算机来说也是天文数字，依然是累计量，而且服务器收到请求后也是先由内核放到缓冲队列中，然后服务器再逐个处理每个请求，拿网络服务器来说，每处理一个请求时，先是调用 accept 系统调用从 socket

队列中获取一个连接 socket，该 socket 包含了与用户的连接信息，然后服务器调用诸如 recv 的系统调用从该连接中读取数据，分析用户的请求是什么。那些实现高并发的服务器也是如此，只是采取了较为先进的 IO 多路复用而显得很快，但依然是一次处理一个。因此只有多核 CPU 才能实现并行。但无论 CPU 有多少个核心，它总会是有限的，总会有任务数多于 CPU 核心数的情况，因此并行是少数的，大部分还是并发。

用户程序运行后，就变成了操作系统眼里的进程，此时的任务都指的是进程，不包括别的。开发人员写出来的程序在运行后就是一个进程，在 CPU 眼里只是一个任务，后来开发人员需要把进程的工作分配给不同的执行流，它们彼此间相互独立，分工协作，共同完成进程的工作，一种方法是使用系统调用 fork，派生出多个子进程，这种方法有个缺点，进程之间并不共享地址空间，它们需要专门的进程间通信机制才能协调工作，因此效率较低。另一种方法较好，在进程内部创建多个独立的执行流，由于是在同一个进程的内部，因此这多个执行流是共享进程的地址空间，相互间通信可以通过一些全局变量来实现，因此较快。这其实就是在进程内实现并发的需求，在进程内的执行流就是线程。实际上子进程之间为了更快地通信，也采取了一种称为共享内存的方式，其原理是操作系统使各进程页表中的页表项指向相同的一块物理内存页，从而实现了部分内存空间的共享。

线程和进程类似，都有独立的程序控制块（PCB）和栈，都是代码的执行流。线程和进程不同的是，进程有独立的虚拟空间，而线程属于进程，进程中的所有线程共享进程的虚拟地址空间。线程实现有两种方式。

一种是在内核态实现，它和进程一样可以上“任务”调度器参与调度，轮换着上 CPU 运行。任务是指可调度的指令流的统称，比如进程和线程，由于此处是在讨论内核态中实现线程，因此此时任务仅指进程和内核线程。能被操作系统任务调度器调度的原因是，线程和进程都有 PCB，这个很重要，内核任务调度器只能调度那些有 PCB 的任务，因为操作系统是通过 PCB 把一个个任务区分开的，PCB 是操作系统为任务颁发的身份证，是操作系统管理任务的一种手段，管理任务的目的就是为了调度。说明一下，任务调度器也是操作系统的一部分，任务调度器使用 PCB 切换任务是操作系统内部协调好的事情。

另外一种是在用户态实现，线程是独立的执行流，只能生存在进程中，无论是在内核态实现还是在用户态实现，这个大前提是不可能变的，内核线程存在于内核进程中，用户线程存在于用户进程中。内核任务调度是通过系统时钟中断周期性发生的，因此每一次任务切换都要先进入内核，这涉及系统调用及上下文保护，成本很高。在线程所属的进程中写一个调度器调度线程，是没办法利用时钟中断的，时钟中断只能切换带有 PCB 的执行流，即进程或内核线程，这种用户态线程没有 PCB，因此没办法通过时钟中断进行用户线程切换。线程是因并发而诞生的，并发就是靠轮换执行实现的，要想轮换执行就要靠调度器的调度，因此进程自己需要实现一个调度器负责其调度。内核态实现的线程，其所使用的调度器是内核的任务调度器，因此相当于 CPU 看得到它，而用户态实现的线程，其调度器是该线程所属进程自己实现的，CPU 根本不知道它的存在，它是和所在的进程混在一起的，CPU 只看得到该进程。这两者的本质区别是内核线程是有自己的 PCB 的，用户进程中的线程并没有自己的 PCB，因为只有具备 PCB 的执行流才能被内核的任务调度器调度，相当于只有具备了操作系统所颁发的“身份证”，操作系统的任务调度器才认得它。为了区分，把用户态实现的线程称为协程，后面有说明其被称为“协”程的原因。以下的用户线程就是指协程。

再讨论一下，为什么通过时钟中断切换任务呢？原因是任务都不想交出宝贵的 CPU 资源，都想一直用，用到运行完为止，时钟中断是不可避免要发生的，通过让时钟中断处理程序调用内核的任务调度器，可强迫当前正在 CPU 上运行的任务下来以进行任务切换，这样所有任务都有机会

运行，不至于出现某个大任务长期霸占 CPU，后面所有小任务无法及时运行的情况，这里的大小指的是运行时间长短。进程自己实现的任务调度器，即协程调度器，无法利用时钟中断（时钟中断属于 ring 0 级，用户进程属于 ring 3 级，可简单地理解为彼此够不着）进行线（协）程调度轮换，所以一定是让协程主动交出 CPU 使用权，也就是在用户线程自己就要协调好使用 CPU 的方式，尽量公平。进程中所有用户线程共同“协作”完成进程的工作，这正是用户态线程称为“协”程的原因。

内核线程和进程还是有弊端的，任务调度是以任务（有 PCB 的才叫任务，如内核线程和进程）为调度单位，阻塞也是如此，因为在操作系统眼里只能看到任务，这对操作系统来说已经是最细粒度了。什么是阻塞呢？就是将正在运行的任务从 CPU 上换下来，直到它的运行条件满足时再把它换上 CPU 运行。总之，不要被阻塞这个名词搞糊涂了，并不是真有一股力量压着按着任务不让运行。为什么阻塞呢？阻塞发生在“即使不阻塞也无法运行”的情况，比如任务 A 执行了读取数据的系统调用，读取的数据来自于低速的磁盘，该磁盘需要响应很久或者当前该磁盘正忙别的读写任务，任务 A 拿不到数据啥也干不了，因为任务 A 中后面的指令都等着要操作这些数据，因此程序的后续运行依赖这些数据（“很少”有后面指令与前面指令无依赖关系的程序，“很少”只是为严谨才加的），所以此时任务 A 不可运行（强行运行的话只能处理空数据，除了取决于程序的健壮性以外，程序结果也必然不正确）。于是操作系统便调用任务调度器把任务 A 换下来，也就是被阻塞，等到磁盘数据准备好了，操作系统再把任务 A 换上 CPU 运行。也许你会问，操作系统怎么知道任务 A 不能继续运行而需要被阻塞呢？因为读取数据的系统调用是操作系统提供的，这个系统调用里有判断是否会引起阻塞的代码，这些代码是用来访问磁盘状态的磁盘驱动程序，驱动程序说磁盘正忙着呢，操作系统就将任务 A 阻塞了。即使磁盘不忙，由于磁盘本身就是低速设备，对于 CPU 来说也慢了 N 个数量级，读取数据的期间对于 CPU 来说无比漫长，而该程序又无法运行，操作系统照样把任务 A 换掉。总之任务调用了出现阻塞的系统调用时，必然会导致任务被挂起。在用户态下实现的线程也会存在这个问题，不仅如此，还会更惨，因为操作系统看不到用户态线程，即使是进程中某个用户线程调用了导致阻塞的系统调用，操作系统也认为是这个用户进程调用的，因此整个进程都会被挂起，也就是说，进程内的一个线程导致整个进程阻塞后，进程内的其他线程也无法运行。但是，用户线程还是有可能规避这种整个进程“连根拔起”的灾难性阻塞，可以在用户态把系统调用再封装一层，在该层内判断系统调用是否会引起阻塞，如果不会就执行，如果会，那么就调用进程自己的线程调度器，调度下一个用户线程上 CPU。如果下一个用户线程也调用了导致阻塞的系统调用，那就再切换，都切换一遍后那就没办法了，所有的用户线程都卡在导致阻塞的系统调用上，该调用还是要调用，该阻塞还是要面对。这虽然麻烦，但确实解决了大部分问题——让进程内的用户线程们尽可能多地运行以尽快地完成进程的工作。用户线程就是用来并发的，既然是并发，这说明进程内的所有用户线程相互之间是独立无依赖的，或者说是开发人员在线程间已经协调好了同步机制，因此可以任意顺序执行而不会出错（同步机制没写错的情况下）。那么，哪些系统调用是阻塞的，哪些又不是呢？一是可以参考操作系统的系统调用接口说明文档，二是凭经验，那些访问低速设备或者外设的系统调用往往都会导致阻塞，而那些不需要外设的系统调用就不会被阻塞，比如获取系统内存容量大小就不需要阻塞，这些数据都是在内存中的，马上就返回。如果某些系统调用不确定呢？那就干脆将其列为会阻塞的系统调用，中间层碰到这种系统调用时就直接换掉当前的用户线程。

对于 CPU 来说，我们的虚拟机就是那个有自己线程调度器的进程，也就是说虚拟机负责用户线程的切换。用户线程调度器的方法是，把执行权交给一个函数，该函数就是用户线程，待用户线程主动交出任务执行权给用户线程调度器后，此调度器保存该线程的上下文到它自己的状态结构中，注意此状态结构相当于 PCB 的作用（但并不是 PCB，PCB 是操作系统颁发给任务的身份

证，是操作系统任务切换的调度单位），然后由用户线程调度器调度下一个用户线程，从它的状态结构中恢复其上下文，把执行权交给它，即换其上 CPU。

尽管用户态线程既实现了并发的优势，又避免陷了入内核的高成本，但它的速度并不如内核线程快，因为用户态线程“们”是共享它们所在的进程的时间片，每个用户线程每次只执行一点点就被换下，而内核线程虽然也处于（内核）进程中，但它是有 PCB 的任务，可以被 CPU 单独调度，因此内核线程是执行整个时间片。这种情况如图 3-4 所示。

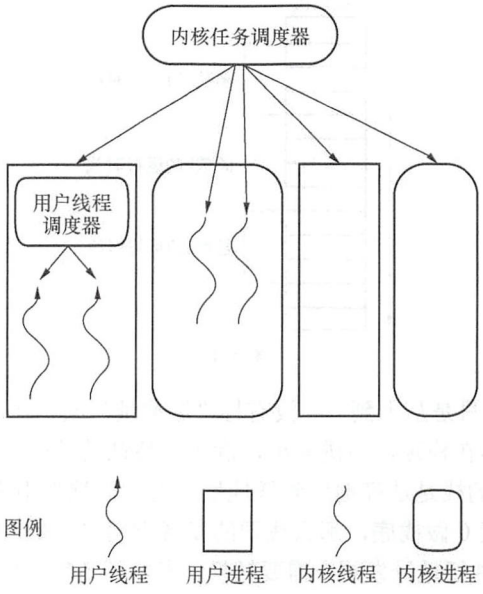


图 3-4

尽管如此，我们还是要实现用户线程，因为内核线程由操作系统实现，我们没办法进入操作系统。

3.15.2 运行时栈

函数不能主动运行，由主调方调用后才能运行。主调方调用函数前需要传递参数，参数是临时使用的，函数运行结束后就没用了，但参数总该有个临时存放的地方，这样函数才能获取到参数，既然用于临时存储，那就临时从栈中分配一块空间吧，编译器从一个“大栈”中分配一小块栈空间给函数（类似列表的切片操作），这样参数就被编译器临时存储到栈中，函数运行时就可以从栈中获取参数。有关这个“大栈”我们在后面会解释。栈都是为函数临时分配的，那就物尽其用，该函数中所有临时存放的都存储到函数的栈中，因此栈还用于存储函数中的局部变量。由于参数和局部变量仅是供函数使用的，而函数的生命周期是短暂的，因此当函数运行结束（也就是函数生命周期结束）后，参数和局部变量就没用了，可以把分配给函数用于存储参数和局部变量的栈回收了，可见这个栈仅是在函数运行时才存在的，因此这个栈称为函数的“运行时”栈。

线程实际上就是一个可以被单独“调度”的函数，注意，不是调用，是调度，只有能被调度器识别的调度单位（执行流）才被称为“可调度”，如果仅是调用的意义，那就纯粹是函数了。我们所实现的用户线程（即协程）既然是函数，那么它必然少不了运行时栈。函数中的参数和局部变量都是单独存储在自己的运行时栈中的，运行时栈相当于作用域，因此不同函数中存在同名的局部变量也不会冲突。一个函数就有一个运行时栈，函数中还可以调用其他函数，那么

到底准备多少个运行时栈才合适呢？不可预计，因为即使数得清调用了多少个函数，也无法避免递归调用，因此就为线程准备一个“大的运行时栈”，就是前面所提到的“大栈”，简称线程栈，线程中若有函数调用，就从线程栈中分配一小块栈空间作为该函数的运行时栈，用完了就回收。由于线程中所有函数共享线程栈，为了避免混乱，每个函数都要记录自己的运行时栈在线程栈中的起始位置，在布局上，各函数的运行时栈是“接壤”的，请读者记住这个特性，以后会用到，如图 3-5 所示。

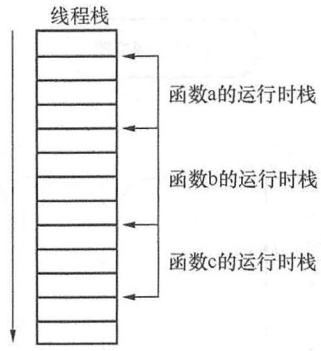


图 3-5

图 3-5 中栈的发展方向只是从上到下，没有标“低地址”或“高地址”，因为栈只是个数据结构，它的特征是最新的数据在栈顶，后进先出，满足此特征的结构就叫栈，与栈顶地址的发展趋势无关。比如硬件 CPU 中的栈是从高地址到低地址发展，而软件中模拟的栈可以用数组来实现，一种方便的做法是数组元素 0 做栈底，那么栈顶的发展方向自然是地址越来越高，如果非要保持和硬件 CPU 中的栈一样，往低地址发展，需要转换，影响了效率。数据结构只是数据存储及操作的思想，实现其思路就可以了。

3.15.3　用户线程的实现

用户线程定义在 object/obj_thread.h 中，代码如下。

stepByStep/c3/c/object/obj_thread.h

```
#ifndef _OBJECT_THREAD_H
#define _OBJECT_THREAD_H
#include "obj_fn.h"

typedef struct objThread {
    ObjHeader objHeader;

    Value* stack;  //运行时栈的栈底
    Value* esp;    //运行时栈的栈顶
    uint32_t stackCapacity;  //栈容量

    Frame* frames;   //调用框架
    uint32_t usedFrameNum;   //已使用的 frame 数量
    uint32_t frameCapacity;  //frame 容量

    //“打开的 upvalue”的链表首节点
    ObjUpvalue* openUpvalues;

```

```
    //当前 thread 的调用者
    struct objThread* caller;

    //导致运行时错误的对象会放在此处,否则为空
    Value errorObj;
} ObjThread;    //线程对象

void prepareFrame(ObjThread* objThread, ObjClosure* objClosure, Value* stackStart);
ObjThread* newObjThread(VM* vm, ObjClosure* objClosure);
void resetThread(ObjThread* objThread, ObjClosure* objClosure);
#endif
```

第 5～24 行定义的 ObjThread 就是线程对象结构，成员 stack 就是线程运行时栈的栈底，esp 是运行时栈的栈顶。

stackCapacity 是线程运行时栈的大小。函数调用离不开运行时栈（也称调用框架、堆栈框架），frames 是函数运行时栈数组，即发生一个函数调用时就从此数组中分配一个运行时栈。也许有读者觉得其实不必为每个函数都准备一个 frame，所有的函数调用都放在同一个栈中就行了，但上节中我们说过，既然所有函数的运行时栈都挤在同一个大栈中（当然是线性排列，并不混插在一起），就需要额外记录各函数运行时栈在大栈中的起始地址。既然也需要找个地方存储起始地址，这就和 frame.stackStart 的作用是一样的，这就是 frame 存在的价值。

usedFrameNum 是已使用的 frame 数量，数组下标从 0 起，因此 usedFrameNum 也是 frames 数组中第一个空闲 frame 的索引。函数调用是不断发生的，总该有个调用顺序，最新的函数调用使用的运行时栈肯定是最新的 frame，因此第 usedFrameNum 个 frame 是当前最新可用的堆栈框架。

frameCapacity 是 frames 数组的容量，即 frame 个数。

openUpvalues 是“打开的 upvalue 链表”的首节点，并不是“open upvalue”链表，因为当 upvalue 的 localVarPtr 指向的局部变量出栈后，该局部变量会存储到 upvalue 的 closedUpvalue 中，因此它同时包括 openUpvalue 和 closedUpvalue，在此相当于文件系统的已打开文件链表，起到缓存的作用。在创建闭包时会用到此缓存。

caller 是指当前线程的调用者，作用是若本线程退出，控制权将回到调用者。

errorObj 是导致运行时错误的对象，否则为空。

下面是线程实现文件，obj_thread.c，代码如下。

stepByStep/c3/c/object/obj_thread.c

```
#include "obj_thread.h"
#include "vm.h"

//为运行函数准备栈桢
void prepareFrame(ObjThread* objThread, ObjClosure* objClosure, Value* stackStart) {
    ASSERT(objThread->frameCapacity > objThread->usedFrameNum, "frame not enough!!");
    //objThread->usedFrameNum 是最新可用的 frame
    Frame* frame = &(objThread->frames[objThread->usedFrameNum++]);

    //thread 中的各个 frame 是共享 thread 的 stack
    //frame 用 frame->stackStart 指向各自 frame 在 thread->stack 中的起始地址
    frame->stackStart = stackStart;
    frame->closure = objClosure;
```

```
14      frame->ip = objClosure->fn->instrStream.datas;
15  }
16
17  //重置 thread
18  void resetThread(ObjThread* objThread, ObjClosure* objClosure) {
19      objThread->esp = objThread->stack;
20      objThread->openUpvalues = NULL;
21      objThread->caller = NULL;
22      objThread->errorObj = VT_TO_VALUE(VT_NULL);
23      objThread->usedFrameNum = 0;
24
25      ASSERT(objClosure != NULL, "objClosure is NULL in function resetThread");
26      prepareFrame(objThread, objClosure, objThread->stack);
27  }
28
29  //新建线程
30  ObjThread* newObjThread(VM* vm, ObjClosure* objClosure) {
31      ASSERT(objClosure != NULL, "objClosure is NULL!");
32
33      Frame* frames = ALLOCATE_ARRAY(vm, Frame, INITIAL_FRAME_NUM);
34
35      //加 1 是为接收者的 slot
36      uint32_t stackCapacity = ceilToPowerOf2(objClosure->fn->maxStackSlotUsedNum + 1);
37      Value* newStack = ALLOCATE_ARRAY(vm, Value, stackCapacity);
38
39      ObjThread* objThread = ALLOCATE(vm, ObjThread);
40      initObjHeader(vm, &objThread->objHeader, OT_THREAD, vm->threadClass);
41
42      objThread->frames = frames;
43      objThread->frameCapacity = INITIAL_FRAME_NUM;
44      objThread->stack = newStack;
45      objThread->stackCapacity = stackCapacity;
46
47      resetThread(objThread, objClosure);
48      return objThread;
49  }
```

第 5 行的函数 prepareFrame 是为线程 objThread 中运行的闭包函数 objClosure 准备运行时栈，起始地址是 stackStart。objClosure 是闭包，其中包括待运行的函数。

usedFrameNum 是最新可用的 frame 索引，第 8 行在线程的数组 frames 中获取最新的 frame 后，使 usedFrameNum 加 1，使 usedFrameNum 始终为最新（第一个）可用的 frame 索引，供下次分配 frame 用。

第 12～14 行初始化 frame 的起始地址为 stackStart，也就是运行闭包函数的运行时栈在线程的栈中的起始地址。待执行的闭包为 objClosure，指令起始地址是闭包中函数的指令流的起始地址，即 objClosure->fn->instrStream.datas。

第 18 行定义的 resetThread 函数用于重置线程，第 19 行将线程栈的栈顶 esp 置为栈底 stack，第 20～23 行就是一般的初始化。最后在第 26 行调用上面介绍的函数 prepareFrame 将线程栈底 objThread->stack 作为闭包 objClosure 中函数的运行时栈的起始地址。

第 30 行定义的 newObjThread 函数用于新建线程，线程中运行的是闭包 objClosure 中的函数。宏 INITIAL_FRAME_NUM 值为 4，定义在 obj_fn.h 中，是线程中默认的 frame 数量。第 33 行为 frames 数组分配内存。第 36 行计算线程栈容量 stackCapacity，这是调用函数 ceilToPowerOf2 计算得出的，实参是 maxStackSlotUsedNum+1，+1 是为了在运行时栈的栈底存储消息的接收者，即类 class 或 this 对象。实例方法的接收者是对象，类方法的接收者是类的 meta 类，以后会细说。第 37 行根据容量 stackCapacity 分配线程栈的内存空间，这里是用宏 ALLOCATE_ARRAY 分配的，可见栈是用数组来实现的。第 39～40 行创建线程并初始化其对象头。再后面就是初始化工作，最后调用 resetThread 函数重置线程，即初始化闭包 objClosure 的运行时栈。

第4章　原生方法及基础实现

4.1 解释器流程（本节源码 stepBystep/c4/a）

终于到了语法分析阶段，前几章为了演示词法分析临时改了程序的流程，现在必须得改回来了，以后也不会再变。从获取源码到执行源码的流程如图 4-1 所示。

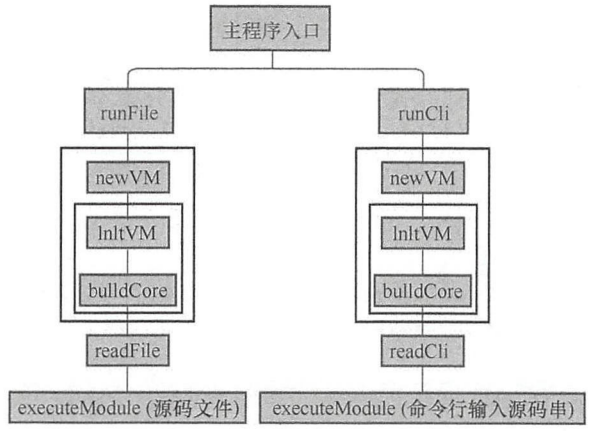


图 4-1

从主程序进入后有两个分支，脚本解析及命令行实时解析。runFile 表示解析脚本文件，这需要调用 newVM 创建一个新的虚拟机，黑色细线方框中的部分表示嵌套调用，在 newVM 中又调用 initVM 初始化虚拟机，初始化包括构建核心模块，因此又调用 buildCore。buildCore 是本章中介绍的函数，用于编译核心源码以提供最原始最核心的支持，这是任何一门语言都要事先提供的功能，即用户未写代码之前已经存在可以使用的代码，用户都要基于此基础来编写代码，否则用户是没办法用这个语言的，比如没有提供输出的话，用户是没办法实现输出功能，不提供列表的话，用户没办法使用 list，也许有读者说："我自己实现 list，不需要系统提供"。那脚本语言的意义何在？不就是为了让用户使用方便吗？核心功能包括基础类如 list 类、map 类、thread 类等，后面会介绍。本小节的工作是将已有代码按照此流程改装。

下面是较上节代码变化的内容，先是 cli.c，代码如下。

stepByStep/c4/a/cli/cli.c

```
...略
 8 //执行脚本文件
 9 static void runFile(const char* path) {
10       const char* lastSlash = strrchr(path, '/');
```

```
11      if (lastSlash != NULL) {
12          char* root = (char*)malloc(lastSlash - path + 2);
13          memcpy(root, path, lastSlash - path + 1);
14          root[lastSlash - path + 1] = '\0';
15          rootDir = root;
16      }
17
18      VM* vm = newVM();
19      const char* sourceCode = readFile(path);
20
21      executeModule(vm, OBJ_TO_VALUE(newObjString(vm, path, strlen(path))), sourceCode);
22  }
…略
```

之前打印 token 的代码全部删掉了。

在脚本语言中代码是以模块为单位，每个模块是单独的名称空间，脚本文件被视为一个模块。第 21 行调用 executeModule 去执行模块，这个函数还没有实现，目前就是个桩函数，即空的，仅起接口的作用。

下面是最新的 core.c。

stepByStep/c4/a/vm/core.c

```
…略
 8 char* rootDir = NULL;   //根目录
 9
10 #define CORE_MODULE VT_TO_VALUE(VT_NULL)
…略
37 //执行模块,目前为空,桩函数
38 VMResult executeModule(VM* vm, Value moduleName, const char* moduleCode) {
39     return VM_RESULT_ERROR;
40 }
41
42 //编译核心模块
43 void buildCore(VM* vm) {
44     //创建核心模块,录入到 vm->allModules
45     ObjModule* coreModule = newObjModule(vm, NULL); //NULL 为核心模块.name
46     mapSet(vm, vm->allModules, CORE_MODULE, OBJ_TO_VALUE(coreModule));
47 }
```

第 10 行定义的宏 CORE_MODULE 用来表示核心模块，其值是 VT_NULL 的 Value 结构形式，后面加载模块时以 CORE_MODULE 作为核心模块的判断依据。

第 38 行定义的 executeModule 函数用于执行模块，脚本文件被视为一个模块，将来脚本的执行也是调用 executeModule。目前是空的，是个桩函数，以后完成相关功能后去填充它。

第 43 行定义的 buildCore 用于编译核心模块。第 45 行通过 newObjModule 创建核心模块 coreModule。allModules 如前在 vm.h 中看到过了，它是个 map，模块是以键值对儿的形式组织的。因此第 46 行调用 mapSet 把核心模块录入到 vm->allModules，key 为 CORE_MODULE，value 为 coreModule 的 Value 结构。我们把所有已加载的模块都放在 allModules 中。此时 buildCore 内容并不完整，以后继续更新。

下面是 vm.c。

stepByStep/c4/a/vm/vm.c

```
#include "vm.h"
#include <stdlib.h>
#include "core.h"

//初始化虚拟机
void initVM(VM* vm) {
    vm->allocatedBytes = 0;
    vm->allObjects = NULL;
    vm->curParser = NULL;
    StringBufferInit(&vm->allMethodNames);
    vm->allModules = newObjMap(vm);
    vm->curParser = NULL;
}

VM* newVM() {
    VM* vm = (VM*)malloc(sizeof(VM));
    if (vm == NULL) {
        MEM_ERROR("allocate VM failed!");
    }
    initVM(vm);
    buildCore(vm);
    return vm;
}
```

initVM 中添加了第 10～12 行 allMethodName、allModules 和 curParser 的初始化，newVM 中添加了 buildCore 的调用。其中 allMethodNames 是脚本中所有类的方法名去重后的集合，后面小节会专门介绍它，在此先完成其初始化。

流程定了，框架也就定了，以后我们的工作就是往里面装东西。

4.2 符号表

符号表就是“名字”表，需要用名字引用的元素都需要把名字存储到符号表中组织起来供随时查询。本节的目的是想让读者对符号管理有个总体上的认识，并不涉及具体，有了全局认识后更容易理解代码。

4.2.1 模块的符号表

模块是 ObjModule 对象，模块也有名字，有名字就要有符号表，模块存储在 VM->allModules 中，它类型是 ObjMap*，我们在 initVM 中已经通过 newObjMap 创建它了。key 为模块的名字，value 为模块的 Value 结构，将来 get 时再将其从 Value 转换为 ObjModule 对象。

4.2.2 类方法的符号表

说明一下，在多数情况下表示类与成员的从属关系时，用->代替“.”，目的是避免误把“.”当成句号，如果是必须用“.”如方法调用，就会用双引号把整个串括起来。

类都有自己的方法，方法存储在 Class 结构中名为 methods 的缓冲区结构中，我们说过，方法属于类的通用数据，通用的部分放在 Class 中存储，一个对象在调用方法时，正常的逻辑是在



对象自己所属的类中寻找方法，也就是在 Class->methods 中寻找方法。如类 A 生成对象 objA，那么在处理方法调用 objA.methodOne 时，解释器应该去 A->methods 中寻找方法 methodOne。函数和方法的区别就是方法只属于类，只能由该类的对象调用，其他方面是一样的，函数由函数声明和函数体组成，那么对应地，方法也应该有方法声明（相当于函数声明，只不过没有存储类型和返回值类型）和方法体（相当于函数体，就是大括号括起的代码块，也就是方法的实现），存储在 Class->methods 中的就是“方法体”。回忆一下，Class->methods 类型是 MethodBuffer，其本质是 Method 结构的数组，数组元素是“方法体”，里面并没有“方法名”，方法名就是字符串，在此结构中并不需要字符串，因为数组元素是用索引来读取的，我们在获取数组中某个方法时只要提供索引就行了。可是别忘了，用户在代码中肯定不会用索引来访问类的方法，肯定用的是方法名，即使它提供了索引，编译器也认为那只是方法名（索引是整数，即整数形式的方法名，由语法分析器负责判断方法名是否合法），不会把它当成 methods 缓冲区的索引。方法名对用户和编译器来说都只是个符号，也就是说，用户是用符号编程的，用户代码在编译器眼里全是符号，用户只关心功能逻辑，如何组织和实现这些符号的功能意义那是编译器自己的事。任何系统都是一套自己协调好的、可自圆其说的方法集合。编译器也是，它在内部把各种符号实现为某种数据结构，编译器根据用户提供的符号去找对应方法的数据结构。对符号的组织需要一个符号表，编译器在内部就是通过这张表建立起符号与之相关数据结构的映射。你看，符号表虽然是编译器内部使用的数据结构，但其更多是侧重于维护用户的符号，即源码。

方法名当然也是符号，它需要用符号表来存储，方法名属于类，尽管我们可以为各个类都建立一个符号表，该符号表专用于存储该类的一切符号，如方法名和域（字段、属性）名，但类很多时，就会存在大量的符号表，维护也麻烦。而且类是一个单独的名称空间，允许不同类中的方法名可以是重复的，如 A 类和 B 类中都可以有同名的方法 M，这种冗余也会浪费空间，因此，最好是只有一个符号表，且相同的方法名只存在一份就行了。一种较好的方案是，我们将所有类的方法名汇总到一起成为一个总的符号表，同名的方法合并为一个，这个总的符号表就是 VM->allmethodNames。注意，VM->allmethodNames 只负责汇总方法名，同名的意义不仅是方法名，还包括形参，“方法名+形参个数”就是方法的签名，签名相当于方法的 ID，实际上就是函数声明（但不包括返回值类型、存储类型）的字符串化，这张大的符号表中存储的就是方法的签名。为什么要用签名来代替方法名呢？因为方法名相同但形参个数不同属于重载，签名中包括了参数个数，签名有效地区分了重载。故为支持重载，我们统一用方法的签名建立方法的符号表。

这种做法相当于把变量的变量名和变量值分开存储了：签名存储在 VM->allmethodNames 中，方法体（即方法的实现）存储在 Class->methods 中。问题是如何建立起签名和方法体的映射使它们成为一体呢。这两个结构本质上是数组，那么用索引来做映射显得容易很多。可是 Class->methods 中元素个数仅是 VM->allmethodNames 元素个数的一部分，肯定不能做一一映射，需要把在 VM->allmethodNames 中的索引转换为对等的 Class->methods 中的索引，不过，转换比较麻烦，还是直接映射简单，也就是让 VM->allmethodNames 中的索引等于 Class->methods 中的索引，条件是只要让 Class->methods 的长度和 VM->allmethodNames 的长度相同，即元素个数相等就可以了。显然 VM->allmethodNames 的长度大于等于各类 Class->methods 的长度，只能“凑数”了，对于类中那些不存在的方法，在 Class->methods 中用空占位填充，就当是空间换时间吧，所有类的 Class->methods 长度都与 VM->allmethodNames 保持一致，这样就可以用方法名的索引来访问不同类中的方法体了。梳理一下，对象调用自己类中的方法时，系统会先从符号表 VM->allmethodNames 获得方法签名的索引，用该索引到具体的类 Class->methods 获取方法体，对象所调用的方法是否存在，在具体类中再做判断。不过正常情况下都是存在的。方法签名的组织



布局如图 4-2 所示。



图 4-2

两个类 ClassA 和 ClassB 所有的方法是从方法 A 到方法 H 一共 8 个，其中方法 H 是 ClassA 和 ClassB 都有的方法，属于同名方法。注意，这里只是说签名一模一样，方法体是否一样都可以，在 VM-> allmethodNames 中的是方法名，方法体是在 Class->methods 中，这里所做的是方法名到方法体的映射。

methods 和 allmethodNames 都是缓冲区结构，数据最终是存储到各缓冲区中的 datas 数组中，如 VM.allmethodNames.datas[index]，用此 index 去获取相应的方法 Class->methods.datas[index]。

有些读者可能觉得这种方法太浪费空间了，其实未必浪费了空间，即使不这么做，也依然节省不了，要么要找个地方存储大符号表中方法名索引到本类方法索引的转换关系，要么就不用大符号表来汇总，单独为每个类存储方法名，重名的冗余照样浪费空间，至少我们这么做获得了速度。

4.2.3　模块变量符号表

在 sparrow 中变量有局部变量和模块变量之分，局部变量是定义在代码块中的变量，如函数中或块参数中的变量，模块变量是指在模块中定义的变量，它位于代码块之外，相当于模块中的全局变量，可以认为，模块变量是模块中最顶（外）层的变量。

是变量就要有名字和值，模块变量的名字和值在它所属的模块中分开存储，类似于方法签名和方法体分开存储。模块对象是 ObjModule，模块变量名存储在 ObjModule->moduleVarName，模块变量的值存储在 ObjModule->moduleVarValue，这两个结构都是缓冲区，本质上也是数组，因此也会用索引来建立名和值之间的映射。也就是说，当识别到一个模块变量时，将其名字存储到 ObjModule->moduleVarName.datas[index]，然后把值存储到 moduleVarvalue 中此索引对应的位置，即 ObjModule->moduleVarValue.datas[index]。模块变量的存储布局如图 4-3 所示。



图 4-3

注意，由于模块变量的名和值分别存储到这两个表中，并且还是用索引来关联，这两个表一定要同步写入以保证索引的一致性，因此最好是原子操作，即要么名和值都写入相应的表，要么都不写入，总之不能一个写了另外一个没写。

4.2.4 局部变量符号表

局部变量称为“局部”的原因是它的活动地盘是局部的，也就是局部作用域。顺理成章地就在它所属的作用域中建立符号表。

作用域在我们这里是用一个被称为编译单元 CompileUnit 的数据结构表示，用成员 localVars 表示局部变量的符号表，当然它也是数组，数组元素的类型是 LocalVar，这是局部变量的结构。我们目前只是介绍存储逻辑，因此就不提前搬出各结构的定义了，以免增加学习成本。

尽管我们是在编译单元中定义局部变量，但并不表示编译单元就是局部作用域，如其名所示，它仅表示一个可以编译的代码单元，比如模块就是一个编译单元，方法和函数也是编译单元，可以这样理解，编译单元是“包含代码”的单元，即可直接执行的部分。在编译单元中用成员 scopeDepth 表示作用域，因此用编译单元管理的变量的作用域类型取决于 scopeDepth，当 scopeDepth 表示模块作用域时，局部变量符号表 localVars 肯定为空，否则表示局部作用域时，如果该作用域中有局部变量，localVars 才不为空。scopeDepth 是作用域的 ID，我们可以为作用域编码，因此作用域可以用数字表示。

再说下局部变量结构 LocalVar，其中包括变量名和作用域，不包括变量值，理解这个很重要，因为之前也说过，局部变量存储在栈中，指的就是“值”存储在栈中，所以没必要在 LocalVar 中额外存储局部变量值。

4.2.5 常量符号表

CompileUnit 之所以称为“编译”单元，是因为它包括指令，我们之前介绍函数对象 ObjFn 时，说明了函数对象并不仅仅表示函数，它用于指代一切指令流单元，编译单元中也要用函数对象作为成员来管理本单元中的指令流。在函数对象中有一个成员 constants，其类型是 ValueBuffer，它就是常量表。

常量是相对于变量而言的术语，指那些不变的量，比如字面量（包括数字、字符串）、类名、函数、模块名和模块变量名。你看，除了函数外，大多数都是和名字相关的，名字是字符串，不可变量。为什么函数也可添加到常量表中呢？原因是为了创建闭包，创建闭包需要以函数为参数，因此为了保证参数（函数）不丢，就放在常量表中。

4.3 方法在运行时栈中的参数

方法是由对象调用的，形式为“对象.方法（arg1,arg2,...）”，方法的运行也需要运行时栈，运行时栈主要用来存储实参和局部变量，这里只说实参——所有实参会存储在一个叫 args 的数组中。这已经是大多数编程语言的约定，一般的参数数组都叫 args。既然参数是存储在运行时栈中，可见，args 是位于栈中的数组。为了在栈中存储各种类型的数值，栈是用 Value 结构的数组模拟的，Value 结构是统一的值类型，任何值都可以转换为 Value，综上两点，args 是位于栈中的 Value 类型的数组，理解这一点很重要。在以上的调用中，对象会存储在参数 args[0]，实参 arg1 会存储在 args[1]中，实参 arg2 会存储在 args[2]中，其他实参以此类推。还有最重要的是，args[0]位于栈底。

函数的调用允许嵌套，比如函数 A 调用函数 B，函数 B 又调用函数 C，函数 A->函数 B->函数 C 就组成了函数的调用链。在函数的调用链中，任意时刻都有个当前运行的函数，也就是调用链的链尾，即函数 C。当前运行的函数称为活动记录，也称为活动框架，当函数 C 执行完毕后，控制权回到函数 B，于是函数 B 就成了活动框架。

args[0]除了存储对象外，还用于存储方法或函数的返回值。返回值是传给函数或方法的主调方，也许你在想，参数被存储在被调方的运行时栈中，主调方是如何拿到参数的呢？不能访问到别人的运行时栈吧。其实之前已经透露过，函数的运行时栈是接壤的，如图 4-4（a）所示。

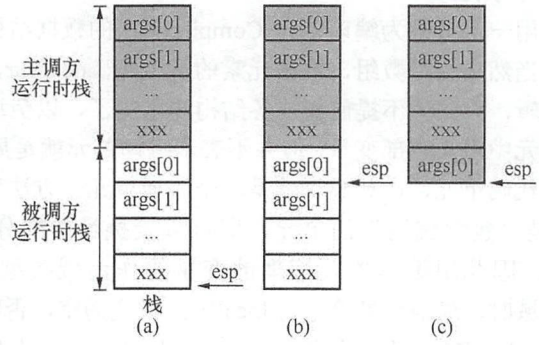


图 4-4

也就是说被调方的栈底和主调方调用被调方前的栈顶是挨着的，被调方将返回值存储到 args[0]后，运行时环境并不会回收被调方 args[0]在栈中的空间，即 esp 的值并未改变，esp 是全局变量，作用于当前的函数，永远是当前活动框架的栈顶，如图 4-4（b）所示。当被调方退出后，原主调方就成了当前活动框架，因此原被调方的 args[0]就成了原主调方的栈顶，如图 4-4（c）所示，那么原主调方访问自己的运行时栈就很正常了，这就是为什么 args[0]一定要位于运行时栈栈底的原因。

4.4 定义模块变量（本节源码 stepByStep/c4/b）

脚本是以模块为组织单位，任何脚本源码文件都会被视为模块。模块变量是模块中最顶层的变量，本节新建文件目录 compiler，其下新建文件 compiler.h 和 compiler.c 实现模块变量的定义，以后语法分析的代码也会添加到 compiler.c。目前 compiler.h 的代码如下。

stepByStep/c4/b/compiler/compiler.h

```
#ifndef _COMPILER_COMPILER_H
#define _COMPILER_COMPILER_H
#include "obj_fn.h"

#define MAX_LOCAL_VAR_NUM 128
#define MAX_UPVALUE_NUM 128
#define MAX_ID_LEN 128    //变量名最大长度

#define MAX_METHOD_NAME_LEN MAX_ID_LEN
#define MAX_ARG_NUM 16

//函数名长度+'('+n 个参数+(n-1)个参数分隔符','+')'
#define MAX_SIGN_LEN MAX_METHOD_NAME_LEN + MAX_ARG_NUM * 2 + 1

#define MAX_FIELD_NUM 128

typedef struct {
    //如果此 upvalue 是直接外层函数的局部变量就置为 true,
```

```
    //否则置为 false
    bool isEnclosingLocalVar;

    //外层函数中局部变量的索引或者外层函数中 upvalue 的索引
    //这取决于 isEnclosingLocalVar 的值
    uint32_t index;
} Upvalue;  //upvalue 结构

typedef struct {
    const char* name;
    uint32_t length;
    int scopeDepth;  //局部变量作用域

//表示本函数中的局部变量是否是其内层函数所引用的 upvalue,
//当其内层函数引用此变量时,由其内层函数来设置此项为 true
    bool isUpvalue;
} LocalVar;     //局部变量

typedef enum {
    SIGN_CONSTRUCT, //构造函数
    SIGN_METHOD,    //普通方法
    SIGN_GETTER,    //getter 方法
    SIGN_SETTER,    //setter 方法
    SIGN_SUBSCRIPT, //getter 形式的下标
    SIGN_SUBSCRIPT_SETTER   //setter 形式的下标
} SignatureType;    //方法的签名

typedef struct {
    SignatureType type; //签名类型
    const char* name;   //签名
    uint32_t length;    //签名长度
    uint32_t argNum;    //参数个数
} Signature;            //签名

typedef struct loop {
    int condStartIndex;   //循环中条件的地址
    int bodyStartIndex;   //循环体起始地址
    int scopeDepth;  //循环中若有 break,告诉它需要退出的作用域深度
    int exitIndex;   //循环条件不满足时跳出循环体的目标地址
    struct loop* enclosingLoop;   //外层循环
} Loop;   //loop 结构

typedef struct {
    ObjString* name;            //类名
    SymbolTable fields;         //类属性符号表
    bool inStatic;              //若当前编译静态方法就为真
    IntBuffer instantMethods;   //实例方法
    IntBuffer staticMethods;    //静态方法
    Signature* signature;       //当前正在编译的签名
} ClassBookKeep;    //用于记录类编译时的信息
```

```
69
70 typedef struct compileUnit CompileUnit;
71
72 int defineModuleVar(VM* vm, ObjModule* objModule, const char* name, uint32_t length,
Value value);
73
74 #endif
```

第 5～16 行定义了一些宏，MAX_LOCAL_VAR_NUM 表示作用域中最大的局部变量个数是 128，MAX_UPVALUE_NUM 表示最大的 upvalue 个数是 128，MAX_ID_LEN 表示标识符中字符数最多是 128，MAX_METHOD_NAME_LEN 表示方法名的最大长度是 128 个字符，MAX_ARG_NUM 表示最多的参数个数是 16。MAX_SIGN_LEN 表示方法签名的长度，签名就是方法声明的字符串化，字符长度包括函数名长度+'('+n 个参数+(n-1)个参数分隔符','+')'。MAX_FIELD_NUM 表示类中域（属性）的最大数量为 128。

第 17～25 行定义的是 Upvalue 结构，表示 upvalue。成员 isEnclosingLocalVar 是个 bool 值，如果此 upvalue 是直接外层函数的局部变量就置为 true，否则置为 false。如果 isEnclosingLocalVar 为 true，成员 index 就表示外层函数中局部变量的索引，否则表示外层函数中 upvalue 的索引。

第 27～35 行定义的是 LocalVar 结构，表示局部变量。成员 name 是局部变量名，length 是变量名的长度，scopeDepth 是局部变量所处的作用域。isUpvalue 是 bool 值，表示本局部变量是否是其内层函数所引用的 upvalue，谁引用谁来设置，因此此项是由其内层函数设置的。局部变量是存储在栈中，因此 LocalVar 中并没有存储局部变量值的成员。

第 37～44 行定义的是 SignatureType 结构，表示签名类型。SIGN_CONSTRUCT 表示构造函数，构造函数就是固定方法名 new。SIGN_METHOD 表示普通的方法，方法形如 XXX(arg1, arg2, ...)。SIGN_GETTER 表示 getter 方法，一般用于读属性，相当于方法 getXXX()，形式为不带括号的普通方法，如 getXXX。SIGN_SETTER 表示 setter，一般用于写属性，相当于方法 setXXX()，形式为 getter 后面再跟上=，如 setXXX=。SIGN_SUBSCRIPT 表示 getter 形式的下标，下标用中括号[]表示，也就是把[]处理为方法，如 list[1]。SIGN_SUBSCRIPT_SETTER 表示 setter 形式的 SIGN_SUBSCRIPT，即在后面加个=，如 list[1] =。所有的 setter 形式在源码中都是被赋值用的，因此=后面会跟一个表达式，如 list[1] =9。

第 46～51 行定义的是 Signature 结构，表示签名，签名就是方法声明的字符串化，如果用函数来理解的话，签名就是函数声明中去掉存储类型和返回值类型后的字符串。成员 type 是签名类型，name 是签名字符串，length 是签名的长度，argNum 是参数个数。

第 53～59 行定义的是 Loop 结构，表示循环，在实现 while 时会用到此结构。成员的意义在注释中写得很清楚，不解释了。

第 61～68 行定义的是 ClassBookKeep 结构，用于记录类编译时的信息，说一下成员 instantMethods 和 staticMethods，它们分别表示实例方法和静态方法的集合，并不是存储方法体，因为其类型是 IntBuffer，即此缓冲区中的元素是 int 型，是数字。ClassBookKeep 结构本身就是辅助记录编译的一些信息，我们用 instantMethods 和 staticMethods 跟踪方法是否在类中重复定义，缓冲区中的元素值是方法在 VM->allMethodNames 中的索引 index，根据方法的类型，每定义一个方法就把方法在 VM->allMethodNames 中的索引写入到 instantMethods 或 staticMethods 中，当然，在写入前要先查看 instantMethods 或 staticMethods 中是否已经有了 index，如果没有说明未定义，这时再插入，否则就报错重复定义。

第 70 行通过 typedef 把“struct compileUnit”定义为 CompileUnit，“struct compileUnit”定义



在 compile.c 中。代码如下。

stepByStep/c4/b/compiler/compiler.c

```
#include "compiler.h"
#include "parser.h"
#include "core.h"
#include <string.h>
#if DEBUG
    #include "debug.h"
#endif

struct compileUnit {
    // 所编译的函数
    ObjFn* fn;

    //作用域中允许的局部变量的个数上限
    LocalVar localVars[MAX_LOCAL_VAR_NUM];

    //已分配的局部变量个数
    uint32_t localVarNum;

    //记录本层函数所引用的 upvalue
    Upvalue upvalues[MAX_UPVALUE_NUM];

  //此项表示当前正在编译的代码所处的作用域
    int scopeDepth;

    //当前使用的 slot 个数
    uint32_t stackSlotNum;

  //当前正在编译的循环层
    Loop* curLoop;

    //当前正编译的类的编译信息
    ClassBookKeep* enclosingClassBK;

    //包含此编译单元的编译单元,即直接外层
    struct compileUnit* enclosingUnit;

    //当前 parser
    Parser* curParser;
};  //编译单元

//在模块 objModule 中定义名为 name,值为 value 的模块变量
int defineModuleVar(VM* vm, ObjModule* objModule,
          const char* name, uint32_t length, Value value) {
    if (length > MAX_ID_LEN) {
          //也许 name 指向的变量名并不以\0 结束,将其从源码串中拷贝出来
          char id[MAX_ID_LEN] = {'\0'};
          memcpy(id, name, length);
```

```
48
49          //本函数可能是在编译源码文件之前调用的,
50          //那时还没有创建 parser, 因此报错要分情况:
51          if (vm->curParser != NULL) {    //编译源码文件
52              COMPILE_ERROR(vm->curParser,
53                    "length of identifier \"%s\" should be no more than %d", id, MAX_ID_LEN);
54          } else {    // 编译源码前调用,比如加载核心模块时会调用本函数
55              MEM_ERROR("length of identifier \"%s\" should be no more than %d", id,
                MAX_ID_LEN);
56          }
57      }
58
59      //从模块变量名中查找变量,若不存在就添加
60      int symbolIndex = getIndexFromSymbolTable(&objModule->moduleVarName, name, length);
61      if (symbolIndex == -1) {
62          //添加变量名
63          symbolIndex = addSymbol(vm, &objModule->moduleVarName, name, length);
64          //添加变量值
65          ValueBufferAdd(vm, &objModule->moduleVarValue, value);
66
67      } else if (VALUE_IS_NUM(objModule->moduleVarValue.datas[symbolIndex])) {
68          //若遇到之前预先声明的模块变量的定义,在此为其赋予正确的值
69          objModule->moduleVarValue.datas[symbolIndex] = value;
70
71      } else {
72          symbolIndex = -1;  //已定义则返回-1,用于判断重定义
73      }
74
75      return symbolIndex;
76  }
```

第 9～39 行定义的 struct compileUnit 正是上面的 CompileUnit，它就是编译单元。之前说过，编译单元就是所编译的代码单位，如函数、方法等独立的指令流都是编译单元。编译单元是指令流，因此少不了函数对象，成员 fn 的作用就是存储其中的指令。数组 localVars 用于记录本编译单元中的局部变量，检测到的局部变量会录入到此数组中。localVarNum 起到游标的作用，用于记录数组 localVars 中下一个可用于存储局部变量的索引，相当于已分配的局部变量个数。upvalues 是本编译单元中引用的 upvalue 的数组。作用域是可以嵌套的，内层更深入，scopeDepth 表示当前正在编译的代码所处的作用域，-1 表示模块作用域，0 表示没有嵌套，即最外层，1 以上表示相应嵌套层。函数和方法这两类编译单元在运行时需要运行时栈，由于后面所用到的是基于栈的虚拟机，栈成了传递参数的地方，很多指令也都要往栈中压入数值或弹出数值，这导致函数或方法所需的运行时栈大小不仅受参数个数和局部变量个数的影响，还要考虑指令对栈的影响，有的指令有压栈操作，故使栈增大，有的指令有弹栈捷足先登，故使栈减小，这里的 stackSlotNum 是用来统计编译单元内的所有指令对栈的最终影响，以供将来为编译单元分配运行时栈做参考。编译单元中的代码少不了循环，循环是可以嵌套的，成员 curLoop 表示当前的循环层。编译类的过程中需要统计类的信息以便于语法检查，方法与函数的区别就是方法有归属类，如果当前编译单元是方法，那么该方法必然属于某个类，成员 enclosingClassBK 用于跟踪当前正编译的类的编译信息。编译单元也是可以嵌套的，成员 enclosingUnit 表示包含当前编译单元的编译单元，即当前

编译单元的直接外层编译单元。之前说过，脚本文件被认为是模块，当在脚本源码中导入一个模块时，即源码中包含关键字 import 时，需要专门为被包含的模块文件创建一个词法分析器。为什么这么做呢？原因是词法分析器与文件句柄相关，分析哪个文件就要调用 open 函数去打开该文件，词法分析器获得该文件句柄，之后的获取字符等文件流操作都是基于此文件句柄，导入另外一个文件就涉及为该文件做词法分析，也就是要 open 该文件，这又是新的文件句柄，一个词法分析器不应该去维护多个文件，因此为每一个文件都创建一个词法分析器单独解析，被包含的文件又可能包含了其他文件，所以在编译一个包含模块导入的源码文件时，编译器中肯定会存在两个以上的 parser。无论编译哪个文件，总该有个当前正在工作的 parser，成员 curParser 就是指向当前的词法分析器。

第 42 行定义的函数 defineModuleVar 用于在模块 objModule 中定义名为 name、值为 value 的模块变量。第 44 行判断模块变量名的长度是否合法，如果长度大于 MAX_ID_LEN 就报错，函数 defineModuleVar 的应用场合有多种，在其被调用时有可能尚未创建词法分析器，有的用来报错的宏需要用到词法分析器，因此第 51～56 行根据是否已有词法分析器选择不同的报错宏。第 50 行调用函数 getIndexFromSymbolTable 从模块中获取模块变量名 name 的索引 symbolIndex，此函数定义在最新的 core.c 中，后面会介绍。如果 symbolIndex 为-1，说明符号 name 未定义，因此第 62～65 行通过函数 addSymbol 和 ValueBufferAdd 将变量名和变量值同步添加到 moduleVarName 和 moduleVarValue。函数 addSymbol 也是定义在 core.c 中，后面会介绍。

第 67～70 行可大有来头，如果 symbolIndex 不等于-1，说明符号 name 已定义，那么进一步判断模块变量 name 的值是否是数字，如果是数字，就证明之前出现过“引用该模块变量时，该模块变量尚未定义”的情况，因为模块变量的作用域是当前模块，相当于当前模块中的全局变量，与引用未定义的局部变量就报错的策略不同，模块变量可以在任意地方定义，万一其定义出现在引用该模块变量的后面呢。也许你觉得有点差异，变量不是先定义再引用吗？那是你的编码风格比较好，我们也提倡先定义再引用，但我们对模块变量给予更大的自由度。再说，类就是按照模块变量来处理的，编译器发现一个类定义时就将其定义为模块变量了，我们允许引用后面定义的类。任何变量一定要有定义才行，模块变量也一样，必须在后期找到其定义才行，否则就报错。如何辨识那些未定义而提前引用的模块变量呢？是这样做的，将来在语法分析时对于被引用的未定义变量，先声明该变量，用该模块变量首次出现时所在的行号作为该模块变量的初始化值，这只是临时的，以后出现该模块变量的定义时再按照正常的方式重新赋值。总之读者要清楚，这里的行号只是个伏笔，如果将来没有遇到该模块变量的定义，我们照样会报错。此时正是处理模块变量的定义，也许正是遇到了之前未定义模块变量的真正定义，所以有可能此时可以解决那个伏笔。在编译阶段值为数字的模块变量就是“伏笔”，因此就需要判断模块变量的值即 moduleVarValue.datas[symbolIndex]是否为数字，如果是数字就证明此时定义的模块变量就是之前引用的未定义模块变量，此时就把该模块变量“正确的值”写入模块变量即 moduleVarValue.datas[symbolIndex]以替换临时的行号，总之，值为数字的模块变量就是之前因为它未定义而被引用留下的伏笔，它是该模块变量未定义的“标签”，期待后面遇到其真正的定义。也许你心比较急，是谁把行号作为未定义模块变量的值呢？是在处理标识符的函数——id 中完成的。此时读者应该会有疑问，模块变量的值为数字很正常吧？定义时其值不能为数字吗？必须能，但是……我们一定要有这样的认识：“赋值并不是在编译时完成的，而是在运行时，任何语言的程序逻辑（比如赋值、判断大小等等）都是在运行时执行的，包括编译型语言也是如此”。我们此时正处于编译阶段，确切地说是语法分析阶段，语法分析时不负责“执行”程序逻辑，只判断脚本代码是否符合语法。变量定义和变量声明不是一回事，“声明”只负责告诉编译器存在这个变量就行了，没有值，默认为 null，那是编译器为简化变量管理的行为。“定义”除了起到声明变量的作

用外，还有一个显式地“用值初始化变量”的过程。初始化即在定义时赋值，因此“定义”=“声明”+“赋值”。模块变量定义的初始化赋值并不是在编译阶段直接写入 moduleVarValue[模块变量 index]，而是分为两步（逻辑上）。

（1）编译阶段：先用 VT_NULL 的 Value 形式声明模块变量的值，即 moduleVarValue[模块变量 index]等于 null。同时生成把变量的值写到 moduleVarValue[模块变量 index]的一系列指令。

（2）运行时阶段：虚拟机在运行时阶段执行上面生成的一系列指令，从而将模块变量的值写入到 moduleVarValue[模块变量 index]，完成赋值。

因此正常情况下，在编译阶段正确定义的模块变量的值 moduleVarValue[模块变量 index]应为 null，运行时阶段才会被赋予正确的值。defineModuleVar 函数是在编译阶段调用的，根据上面第一步，正常情况下参数 value 的值就是 VT_NULL 的 Value 结构，对于脚本来说就是 null。除非是上面的“伏笔”情况：编译器在“编译阶段”发现用户代码引用的模块变量未定义时，先用行号即类型为 VT_NUM 来定义这个模块变量，那时 value 的类型才会是数字。此时你清楚了，这里形参 value 不是模块变量真正的值，value 还是 null，而这就是正常的情况，真正赋值是发生在运行时，除非是之前遇到了未定义的情况，否则编译阶段中模块变量值的类型不会是数字，因此数字类型就是未定义模块的标签。既然正常情况下值是 null，并不是模块变量真正的值，那为什么此处还要用 null 覆盖行号呢？这纯粹是为了替换数字类型的行号以去掉该模块变量未定义的标签。这是一遍的编译器，在扫描完整个源码后我们才知道到底定义了哪些模块变量，为了确定所有引用的未定义模块变量在后面都定义过，我们在扫描完源码后，即编译阶段结束时且未运行虚拟机之前，还会去检查 moduleVarValue 中那些值类型为 VT_NUM 的模块变量，如果存在这类变量，说明一直没遇到其真正的定义，这时再报错。

如果流程执行到了第 71 行，说明模块变量重复定义，在第 75 行返回-1。

4.5 原生方法（本节源码 stepByStep/c4/b）

我们是用 C 语言开发脚本语言，脚本语言中的一些方法，如果是在内部用 C 语言直接实现的，那么该方法就是原生方法。

4.5.1 定义裸类

一般的类都有归属类，也就是有元信息类即 meta-class，裸类是没有归属类的类，我们要基于它实现很多类定义，继续在 class.c 中补充代码。

stepByStep/c4/b/object/class.c

```
...略
51 //新建一个裸类
52 Class* newRawClass(VM* vm, const char* name, uint32_t fieldNum) {
53     Class* class = ALLOCATE(vm, Class);
54
55     //裸类没有元类
56     initObjHeader(vm, &class->objHeader, OT_CLASS, NULL);
57     class->name = newObjString(vm, name, strlen(name));
58     class->fieldNum = fieldNum;
59     class->superClass = NULL;    //默认没有基类
60     MethodBufferInit(&class->methods);
61
62     return class;
```

```
63 }
64
65 //数字等 Value 也被视为对象,因此参数为 Value.获得对象 obj 所属的类
66 inline Class* getClassOfObj(VM* vm, Value object) {
67     switch (object.type) {
68         case VT_NULL:
69             return vm->nullClass;
70         case VT_FALSE:
71         case VT_TRUE:
72             return vm->boolClass;
73         case VT_NUM:
74             return vm->numClass;
75         case VT_OBJ:
76             return VALUE_TO_OBJ(object)->class;
77         default:
78           NOT_REACHED();
79     }
80     return NULL;
81 }
```

第 52 行定义的 newRawClass 函数用于新建域个数为 fieldNum、名字为 name 的裸类。在第 53 行申请内存后，第 54 行为其初始化对象头，第 4 个实参为 NULL，也就是说对象头中 class 指针指空，没有归属类，这就是裸类。后面的初始化比较容易。

面向对象语言中任何数据都是对象，是对象就会有归属类，归属类存在的目的是为对象提供可调用的方法。判断一个对象的归属类，是看其对象头中 class 指针。如果对象并不是用对象结构如 ObjXX 定义的话，也就是没有对象头，因此无法通过对象头中 class 指针找到自己的归属类，那该如何使用类方法呢？比如数字也是个对象，那完全没必要为把数字封装为一个对象故意定义一个复杂的结构，数字很多，极大地浪费了空间。对于这种对象，必须为其找一个归属类，这样才能使用相应类中的方法。这就是我们提前在 VM 结构中添加 boolClass 和 numClass 等类指针的原因，后面我们会为这两个指针赋值正确的类，我们可以根据数据的类型来判断数据的归属类，比如若是数字，就返回 VM->numClass 的归属类。

第 66 行定义的 getClassOfObj 函数用于获取形参 object 的归属类，第 68～74 行就是根据数据 object 的类型来返回相应的归属类。第 75 行判断数据类型若是 VT_OBJ，说明是用 ObjXXX 定义的对象，包含对象头，因此返回对象头中 class 指针指向的类。

4.5.2 定义返回值与方法绑定的宏

本节将在 C 代码中实现原生方法，原生方法定义在 core.c 中，代码有点长，部分代码如下。

stepByStep/c4/b/vm/core.c (1/5)

```
...略
 7 #include "compiler.h"
 8
 9 char* rootDir = NULL;   //根目录
10
11 #define CORE_MODULE  VT_TO_VALUE(VT_NULL)
12
13 //返回值类型是 Value 类型,且是放在 args[0], args 是 Value 数组
14 //RET_VALUE 的参数就是 Value 类型,无须转换直接赋值
```

```
15 //它是后面"RET_其他类型"的基础
16 #define RET_VALUE(value)\
17     do {\
18         args[0] = value;\
19         return true;\
20     } while(0);
21
22 //将 obj 转换为 Value 后作为返回值
23 #define RET_OBJ(objPtr) RET_VALUE(OBJ_TO_VALUE(objPtr))
24
25 //将 bool 值转为 Value 后作为返回值
26 #define RET_BOOL(boolean) RET_VALUE(BOOL_TO_VALUE(boolean))
27 #define RET_NUM(num) RET_VALUE(NUM_TO_VALUE(num))
28 #define RET_NULL RET_VALUE(VT_TO_VALUE(VT_NULL))
29 #define RET_TRUE RET_VALUE(VT_TO_VALUE(VT_TRUE))
30 #define RET_FALSE RET_VALUE(VT_TO_VALUE(VT_FALSE))
31
32 //设置线程报错
33 #define SET_ERROR_FALSE(vmPtr, errMsg) \
34     do {\
35         vmPtr->curThread->errorObj = \
36           OBJ_TO_VALUE(newObjString(vmPtr, errMsg, strlen(errMsg)));\
37         return false;\
38     } while(0);
39
40 //绑定方法 func 到 classPtr 指向的类
41 #define PRIM_METHOD_BIND(classPtr, methodName, func) {\
42     uint32_t length = strlen(methodName);\
43     int globalIdx = getIndexFromSymbolTable(&vm->allMethodNames, methodName, length);\
44     if (globalIdx == -1) {\
45         globalIdx = addSymbol(vm, &vm->allMethodNames, methodName, length);\
46     }\
47     Method method;\
48     method.type = MT_PRIMITIVE;\
49     method.primFn = func;\
50     bindMethod(vm, classPtr, (uint32_t)globalIdx, method);\
51 }
…略
```

栈是 Value 结构数组，返回值是存储在栈中，因此任何类型的返回值都要转换为 Value 结构。前面说过，返回值是存储到 args[0]中，为便于把返回值转换为 Value 后存储到 args[0]，预先定义了一些宏。

第 16 行的 RET_VALUE 用于把参数 value 存储到 args[0]，然后返回 true。由于宏是用在原生方法中，也就是 C 代码中，而我们的原生方法中需要返回值，此 true 就是原生方法的返回值，与用户脚本无关。此返回值的作用和虚拟机有关，虚拟机在回收函数或方法的参数时会根据此返回值为 true 或 false 执行不同的回收策略，返回值为 false 还涉及线程切换，以后实现虚拟机时再详述。

RET_VALUE 是后面所有返回值类型 RET_XXX 的基础，它们把参数统一转换为 Value 类型后会调用 RET_VALUE 存储返回值。

第 23 行定义的宏 RET_OBJ 用于返回对象。第 26～30 行用于一般数据。

以上的宏都是返回 true，下面的 SET_ERROR_FALSE 返回的是 false。SET_ERROR_FALSE 用于设置线程报错，线程中的 errorObj 成员用于存储报错信息，这里是把报错信息 errMsg 封装为字符串对象存储到 errorObj 中，最后返回 false。返回 false 是为了告诉虚拟机应该切换线程了，因为当前线程已经出错了。以后实现虚拟机时就清楚了。

第 41 行定义的宏 PRIM_METHOD_BIND 用于绑定方法 func 到 classPtr 指向的类，其中 methodName 是在脚本中使用的方法，绑定后，脚本中类 classPtr 的方法 methodName 对应的原生方法是 func，即脚本中调用 methodName 方法就是调用 C 中的 func 函数。第 43 行调用 getIndexFromSymbolTable 从 allMethodNames 中获得方法 methodName 的索引，getIndexFromSymbolTable 定义在后面，一会介绍。如果返回值 globalIdx 为-1，说明该符号并不存在，那么第 45 行就调用 addSymbol 将其添加到 allMethodNames，addSymbol 用于添加符号，也是在后面定义的，其返回值便是符号 methodName 在 allMethodNames 中的索引。第 47～49 行生成方法 method，方法类型为 MT_PRIMITIVE，即原生方法，方法指针 primFn 为参数 func，func 是我们的原生实现。最后调用 bindMethod 把 method 绑定到 classPtr 指向的类，bindMethod 在后面定义，提前说下，这里的参数 globalIdx 在 bindMethod 中用于在类的 methods 数组中索引待绑定方法体的位置，就是之前咱们所说的通过索引把 VM->allMethodNames 中的方法名与 Class->methods 中的方法体一一映射。

第 51～77 行之间的是之前介绍过的部分，略过。

4.5.3 定义原生方法

本节代码中第 79 行以下便是原生方法的实现函数，它们就是宏 PRIM_METHOD_BIND 的形参 func 所指的函数。原生的英文单词是 primitive，原生方法的函数命名规则由 3 部分组成：prim+类 XXX+操作 XXX，从方法名上就能看出，类 XXX 中的操作 XXX 是原生方法。每个类中都有自己的一套原生方法，函数名中“类 XXX”表示此方法是属于类 XXX，与其他类无关。

以下定义的函数，为突显函数的功能意义，在其上的注释中会加上应用场景，以冒号分隔。代码如下。

stepByStep/c4/b/vm/core.c (2/5)

```
78 //!object: object 取反,结果为 false
79 static bool primObjectNot(VM* vm UNUSED, Value* args) {
80       RET_VALUE(VT_TO_VALUE(VT_FALSE));
81 }
82
83 //args[0] == args[1]: 返回 object 是否相等
84 static bool primObjectEqual(VM* vm UNUSED, Value* args) {
85     Value boolValue = BOOL_TO_VALUE(valueIsEqual(args[0], args[1]));
86     RET_VALUE(boolValue);
87 }
88
89 //args[0] != args[1]: 返回 object 是否不等
90 static bool primObjectNotEqual(VM* vm UNUSED, Value* args) {
91     Value boolValue = BOOL_TO_VALUE(!valueIsEqual(args[0], args[1]));
92     RET_VALUE(boolValue);
93 }
94
95 //args[0] is args[1]:类 args[0]是否为类 args[1]的子类
```

```
 96 static bool primObjectIs(VM* vm, Value* args) {
 97     //args[1]必须是 class
 98     if (!VALUE_IS_CLASS(args[1])) {
 99         RUN_ERROR("argument must be class!");
100     }
101
102     Class* thisClass = getClassOfObj(vm, args[0]);
103     Class* baseClass = (Class*)(args[1].objHeader);
104
105     //有可能是多级继承,因此自下而上遍历基类链
106     while (baseClass != NULL) {
107
108         //在某一级基类找到匹配，就设置返回值为 VT_TRUE 并返回
109         if (thisClass == baseClass) {
110             RET_VALUE(VT_TO_VALUE(VT_TRUE));
111         }
112         baseClass = baseClass->superClass;
113     }
114
115     //若未找到基类,说明不具备 is_a 关系
116     RET_VALUE(VT_TO_VALUE(VT_FALSE));
117 }
118
119 //args[0].tostring: 返回 args[0]所属 class 的名字
120 static bool primObjectToString(VM* vm UNUSED, Value* args) {
121     Class* class = args[0].objHeader->class;
122     Value nameValue = OBJ_TO_VALUE(class->name);
123     RET_VALUE(nameValue);
124 }
125
126 //args[0].type:返回对象 args[0]的类
127 static bool primObjectType(VM* vm, Value* args) {
128     Class* class = getClassOfObj(vm, args[0]);
129     RET_OBJ(class);
130 }
131
132 //args[0].name: 返回类名
133 static bool primClassName(VM* vm UNUSED, Value* args) {
134     RET_OBJ(VALUE_TO_CLASS(args[0])->name);
135 }
136
137 //args[0].supertype: 返回 args[0]的基类
138 static bool primClassSupertype(VM* vm UNUSED, Value* args) {
139     Class* class = VALUE_TO_CLASS(args[0]);
140     if (class->superClass != NULL) {
141         RET_OBJ(class->superClass);
142     }
143     RET_VALUE(VT_TO_VALUE(VT_NULL));
144 }
145
```

```
146 //args[0].toString: 返回类名
147 static bool primClassToString(VM* vm UNUSED, Value* args) {
148     RET_OBJ(VALUE_TO_CLASS(args[0])->name);
149 }
150
151 //args[0].same(args[1], args[2]): 返回 args[1]和 args[2]是否相等
152 static bool primObjectmetaSame(VM* vm UNUSED, Value* args) {
153     Value boolValue = BOOL_TO_VALUE(valueIsEqual(args[1], args[2]));
154     RET_VALUE(boolValue);
155 }
...略
```

第 79 行定义的函数 primObjectNot 用于对象取反，其应用场景如注释!object。按照命名规则，这是用于处理 object 类中的 Not 取非操作，即对象取反时执行的方法。处理逻辑是调用 VT_TO_VALUE 把 VT_FALSE 封装为 Value 形式，然后调用 RET_VALUE 设置返回值，即一律返回 false。这里的 VT_FALSE 就是脚本中的 false，以后会看到相应的逻辑。注意，此 object 是 object 类，object 是所有类的根，在 object 类中定义的方法可被所有对象调用。也许有读者有疑问，bool 值也是对象，对 false 取反结果应该为 true。没错，但要清楚的是，object 类中的方法是属于全体对象，换句话说，如果某子类对象调用了子类中没有的方法，最终会上溯到 object 类中查找方法，如果子类中有相应的方法，除非代码中显式调用，否则是不会调用 object 类中的方法的。我们以后会定义这个子类——false 类，专门在其类中准备了取反的方法。再强调一下，函数名中“类 XXX”表示此方法是属于类 XXX，与其他类无关。

第 84 行和第 90 行定义的 primObjectEqual 和 primObjectNotEqual 分别用于判断两个对象是否相等和是否不等，即 primObjectEqual 是在脚本源码中出现 args[0] == args[1]时才会调用的，primObjectNotEqual 是在脚本源码中出现 args[0] != args[1]时才会调用的，处理逻辑都很简单。

后面其余的原生方法（以后会陆续补充）道理类似，而且都比较简单，读者自己看吧，先有个印象即可，在后面虚拟机中应用的时候就会融会贯通了。

4.5.4 符号表操作

本节实现的功能依然在 core.c 中，符号表操作包括查询和添加，下面是 core.c 的第 3 部分，代码如下。

stepByStep/c4/b/vm/core.c (3/5)

```
...略
162 //table 中查找符号 symbol，找到后返回索引，否则返回-1
163 int getIndexFromSymbolTable(SymbolTable* table, const char* symbol, uint32_t length)
{
164     ASSERT(length != 0, "length of symbol is 0!");
165     uint32_t index = 0;
166     while (index < table->count) {
167         if (length == table->datas[index].length &&
168                 memcmp(table->datas[index].str, symbol, length) == 0) {
169             return index;
170         }
171         index++;
172     }
173     return -1;
```

```
174 }
175
176 //往 table 中添加符号 symbol,返回其索引
177 int addSymbol(VM* vm, SymbolTable* table, const char* symbol, uint32_t length) {
178     ASSERT(length != 0, "length of symbol is 0!");
179     String string;
180     string.str = ALLOCATE_ARRAY(vm, char, length + 1);
181     memcpy(string.str, symbol, length);
182     string.str[length] = '\0';
183     string.length = length;
184     StringBufferAdd(vm, table, string);
185     return table->count - 1;
186 }
```

第 163 行定义的 getIndexFromSymbolTable 用于从符号表 table 中查找符号 symbol，找到后返回索引，否则返回-1。SymbolTable 是 StringBuffer 的别名，之前定义过它。符号查询是个遍历符号表的过程，符号表 table 的长度是 table->count，通过对每个元素比较符号字符串及长度，若找到就返回索引 index，遍历结束后若没有找到就返回-1。

第 171 行定义的函数 addSymbol 用于往 table 中添加符号 symbol，然后返回其索引。符号表 table 中的元素类型是 String，第 179 行生成 String 变量 string，用来存储符号 symbol，第 180～181 行为 string.str 申请空间存储符号 symbol 的空间，然后复制 symbol 到 string.str。第 182 行添加结束字符\0。第 183 行更新 string.length 的值为符号 symbol 的长度 length，将 symbol 封装为 string 后，第 184 行调用 StringBufferAdd 将 string 写入到符号表 table。StringBufferAdd 函数会使元素个数 count 加 1，count 是下一个元素的索引，因此第 185 行将 count 减 1 后作为符号 symbol 的索引返回。

4.5.5　定义类、绑定方法、绑定基类

在 core.c 中还有一些类操作，代码如下。

stepByStep/c4/b/vm/core.c (4/5)

```
187
188 //定义类
189 static Class* defineClass(VM* vm, ObjModule* objModule, const char* name) {
190     //1. 先创建类
191     Class* class = newRawClass(vm, name, 0);
192
193     //2. 把类作为普通变量在模块中定义
194     defineModuleVar(vm, objModule, name, strlen(name), OBJ_TO_VALUE(class));
195     return class;
196 }
197
198 //使 class->methods[index]=method
199 void bindMethod(VM* vm, Class* class, uint32_t index, Method method) {
200     if (index >= class->methods.count) {
201         Method emptyPad = {MT_NONE, {0}};
202         MethodBufferFillWrite(vm, &class->methods, emptyPad, index - class->methods.
            count + 1);
203     }
204     class->methods.datas[index] = method;
```

```
205 }
206
207 //绑定基类
208 void bindSuperClass(VM* vm, Class* subClass, Class* superClass) {
209     subClass->superClass = superClass;
210
211     //继承基类属性数
212     subClass->fieldNum += superClass->fieldNum;
213
214     //继承基类方法
215     uint32_t idx = 0;
216     while (idx < superClass->methods.count) {
217         bindMethod(vm, subClass, idx, superClass->methods.datas[idx]);
218         idx++;
219     }
220 }
```

第 189 行定义的函数 defineClass 用于定义类。第 191 行调用 newRawClass 创建裸类，属性（即域）的个数是 0，原因是这是一遍的编译器，编译器在脚本中遇到关键字 class 时就在内部为其定义一个类，属性是在类体中，定义类时属性个数还不知道，编译完类后才知道，后面会在编译结束后更新属性的个数。第 194 行调用函数 defineModuleVar 把类作为模块变量定义在模块中。defineModuleVar 是在其他文件中实现的，用于在模块中定义模块变量，后面小节会说到。

第 199 行定义的 bindMethod 函数用于绑定方法，所谓绑定方法就是把方法添加到类的 methods 数组中。索引 index 是插入到数组中的位置。之前讨论为使各类自己的 methods 数组和 VM->allmethodNames 保持同样长度，用空占位填充，它被应用到 bindMethod 方法中了。第 200～203 行判断若 index 大于方法数组 methods 的长度，就创建空填充块 emptyPad，类型为 MT_NONE，值为 0。通过函数 MethodBufferFillWrite 连续写入 index - class->methods.count + 1 个 emptyPad 以实现填充占位，最后第 204 行把 method 写入到 index 的位置。

习惯称继承的类为基类而不是父类，因为父类只有 1 个，而继承的类可以有多个，因此称为父类不符合逻辑。

第 208 行定义的函数 bindSuperClass 用于绑定 superClass 为 subClass 的基类。绑定基类就是继承基类的属性个数和方法，这是通过复制来实现的。首先在第 209 行将子类的 superClass 指针指向基类 superClass，然后第 212 行在子类属性数中追回基类属性个数，第 215～219 行循环调用 bindMethod 绑定所有基类中的方法到子类中。

4.6 元类及实现（本节源码 stepByStep/c4/b）

4.6.1 meta-class 类、class 类、object 类

我们是用 struct 结构体来实现各种对象的，为什么结构体能够成为对象呢？因为它包含了一个对象头结构，对象是具有类型的实例，对象头中有一个 class 指针，它指向对象所属的类，任何包括对象头的 struct 结构都可以认为是一个对象，甚至包括类的结构——Class 结构，Class 被称为类对象。

学过多门面向对象语言的读者肯定有体会，对象调用方法时，有的语言中称这为向对象发送消息，并且把对象称为消息的接收者。通过之前的介绍，我们知道对象调用方法时，解析器（确切地说是运行时环境——虚拟机）是在对象所属的类中寻找调用的方法，它是通过对象结构体中

对象头的 class 指针找到其所属类 Class，然后在 Class->methods 中找到相应的方法体，进而执行方法体，实现了方法的调用。再深入一步，类中的方法大致上分两种，一种是实例方法，也称对象方法，只能由对象调用，另一种是类方法，也称静态方法，因为它不属于任何一个对象，属于所有对象共有的方法，因此可以由对象调用，也可以通过类名来调用。对象调用方法的逻辑读者都明白，那通过类调用类方法是怎么回事呢？这就引出了 meta-class。

我们说过了，只要包含对象头的 struct 结构体就可以称为对象，在 Class 结构中也包含对象头，那么 Class 也有自己所属的类了？是的，Class 的类，即类的类，是 meta-class，meta-class 是一个类对象的类。由于 meta-class 是在运行时调用的，即虚拟机调用的，并不暴露给用户，因此很多用户并不知道它是什么。它有很多名字，有的称之为原始类，有的直译为元信息类。meta-class 称为元信息类是有道理的，元信息读者都清楚，就是用来描述信息的信息，那么元信息类就是描述类信息的类，描述类的什么呢？肯定是方法。不要被 meta-class 搞糊涂了，它只是个概念，只起到元信息的作用，并不是有一个名叫 meta-class 的类。用对象来理解它就清楚了。

（1）类描述了对象可调用的方法等，对象的元信息就放在对象的所属类，“对象的类”就是对象的元信息。

（2）meta-class 描述了类可以调用的方法等，类的元信息就放在类的所属类，“类的类”称为类的元信息。

对象的类和类的类都是元信息，对象的类就叫类，类的类就叫 meta-class。

meta-class 对类来说是至关重要的，因为它存储着一个类的所有类方法，即静态方法。因为每个类的类方法可以是独一无二的，所以每个类都要有单独的 meta-class。当通过类调用类方法时，运行时系统会从类结构中对象头的 class 指针所指的 meta-class 中获取类方法，meta-class 是元信息类的统称，类名不限制，也是用 Class 结构定义的，类结构是通用的，因此在 sparrow 中，是在 meta-class->methods 中寻找方法体。meta-class 存在的意义就是可以通过类调用类方法，让类成为消息的接收者。总结如下：

（1）对象调用实例方法，也就是向一个对象发送消息时，运行时系统会在对象所属类的方法集合中查找方法。

（2）类调用类方法，也就是向一个类发送消息时，运行时系统会在类的 meta-class 的方法集合中查找方法。

既然 meta-class 也是 Class 结构，那么也包含对象头，那么对象头中的 class 指针指向哪里？即 meta-class 的元信息类是谁？这没有统一，各类语言有自己的处理方式，我们是这样处理的：所有 meta-class 都指向同一个类——class，类名就是 class，这里为了和 class 指针区分，用双引号括起的 class 就表示类名。我们会专门定义 class，它是所有 meta-class 的类，即元信息类。既然 meta-class 的对象头指向了 class，我们也可以向 meta-class 发送消息。由于 class 也是 Class 结构体类型，它在结构上与普通的类没有区别，因此它也包含对象头，那么它的 class 指针指向哪里？肯定不能再引入另一个类了，否则就无穷无尽没头了。我们让它的 class 指针指向自己，这样就形成完美的闭环。

类都有自己的基类，也就是在定义类时继承一个类，通常情况下类的基类是显式定义的，在 sparrow 中用关键字<来表示继承。如果没有显式定义基类，那么编译器会把 object 类作为其基类。注意，object 虽然是对象的意思，但这里表示的是一个叫 object 的类，与对象无关。object 类和普通类的结构相同，同样包括对象头，因此也有元信息类，即 meta-class，在 sparrow 中，它的 meta-class 类名为 object metaClass，这也是我们自定义的类。不仅是 sparrow，其他语言中也有这个 object 类，只不过有的是叫 root 类，意思都是一样的，都表示老祖宗类，在 sparrow 中，除 meta-class 外的任何类都是它的子类，包括我们刚刚说过的 class 也是它的子类。那么 meta-class 的基类是谁

仅供非商业用途或交流学习使用

呢？是 class 类。meta-class 类的基类和元类都是 class。有点晕，图 4-5 梳理了这些关系：

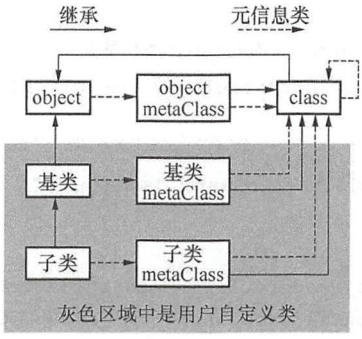


图 4-5

后面我们会按照上图中逻辑关系创建各类。

4.6.2 创建元类，绑定类方法

核心模块包括原生方法和一些脚本，脚本以后再说，目前已经定义了一小部分原生方法，本节完成两件事：一是绑定这些原生方法，二是建立 meta-class 类。继续看 core.c，代码如下。

stepByStep/c4/b/vm/core.c (5/5)

```
221
222 //编译核心模块
223 void buildCore(VM* vm) {
224
225     //核心模块不需要名字,模块也允许名字为空
226     ObjModule* coreModule = newObjModule(vm, NULL);
227
228     //创建核心模块,录入到 vm->allModules
229     mapSet(vm, vm->allModules, CORE_MODULE, OBJ_TO_VALUE(coreModule));
230
231     //创建 object 类并绑定方法
232     vm->objectClass = defineClass(vm, coreModule, "object");
233     PRIM_METHOD_BIND(vm->objectClass, "!", primObjectNot);
234     PRIM_METHOD_BIND(vm->objectClass, "==(_)", primObjectEqual);
235     PRIM_METHOD_BIND(vm->objectClass, "!=(_)", primObjectNotEqual);
236     PRIM_METHOD_BIND(vm->objectClass, "is(_)", primObjectIs);
237     PRIM_METHOD_BIND(vm->objectClass, "toString", primObjectToString);
238     PRIM_METHOD_BIND(vm->objectClass, "type", primObjectType);
239
240     //定义 classOfClass 类,它是所有 meta 类的 meta 类和基类
241     vm->classOfClass = defineClass(vm, coreModule, "class");
242
243     //objectClass 是任何类的基类
244     bindSuperClass(vm, vm->classOfClass, vm->objectClass);
245
246     PRIM_METHOD_BIND(vm->classOfClass, "name", primClassName);
247     PRIM_METHOD_BIND(vm->classOfClass, "supertype", primClassSupertype);
248     PRIM_METHOD_BIND(vm->classOfClass, "toString", primClassToString);
249
```

仅供非商业用途或交流学习使用


```
250     //定义 object 类的元信息类 objectMetaclass,它无须挂载到 vm
251     Class* objectMetaclass = defineClass(vm, coreModule, "objectMeta");
252
253     //classOfClass 类是所有 meta 类的 meta 类和基类
254     bindSuperClass(vm, objectMetaclass, vm->classOfClass);
255
256     //类型比较
257     PRIM_METHOD_BIND(objectMetaclass, "same(_,_)", primObjectmetaSame);
258
259     //绑定各自的 meta 类
260     vm->objectClass->objHeader.class = objectMetaclass;
261     objectMetaclass->objHeader.class = vm->classOfClass;
262     vm->classOfClass->objHeader.class = vm->classOfClass; //元信息类回路,meta 类终点
263 }
```

第 223 行定义的函数 buildCore 用于编译核心模块，实现原生方法的绑定和建立类关系，以下代码读者可以参照图 4-5 来熟悉代码。第 226 行通过 newObjModule 创建核心模块，核心模块的名字为 NULL。第 229 行通过 mapSet 把核心模块注册到 vm->allModules。allModules 是 map，核心模块在 allModules 中的 key 为 CORE_MODULE，value 为 coreModule 的 Value 形式。第 232～238 行创建 object 类并挂到 vm->objectClass，随后绑定原生方法到 objectClass，都是之前介绍的方法和宏，希望读者能贯穿起来。第 241 行定义类 class，它是所有 meta-class 的 meta-class，即类的类，把 class 类挂在 VM->classOfClass。第 244 行调用 bindSuperClass 作为 objectClass 的子类。接下来第 246～248 行绑定原生方法到 classOfClass。第 251 行为 object 类创建元类——objectMeta 类，VM 中并没有其挂载点，因此直接赋值给 objectMetaclass 指针变量。classOfClass 类是所有 meta-class 类的基类，第 254 行调用 bindSuperClass 使 classOfClass 成为 objectMetaclass 的基类。第 257 行在 objectMetaclass 中绑定方法后，第 259～262 行按照类关系图 4-5 绑定元类关系。

4.7 加载模块（本节源码 stepByStep/c4/c）

脚本文件被认为是一个模块，加载模块是从编译到执行的过程，目前先完成模块加载的准备工作。

先在 compiler.c 中添加个桩函数，代码如下。

stepByStep/c4/c/compiler/compiler.c

```
…略
78 //编译模块（目前是桩函数）
79 ObjFn* compileModule(VM* vm, ObjModule* objModule, const char* moduleCode) {
80     ;
81 }
…略
```

compileModule 函数用于编译模块，其实现涉及很多未完成的东西，目前就先把架子搭起来，函数体为空。

core.c 中有一些更新，代码如下。

stepByStep/c4/c/vm/core.c

```
…略
157 //从 modules 中获取名为 moduleName 的模块
```

仅供非商业用途或交流学习使用


```
158 static ObjModule* getModule(VM* vm, Value moduleName) {
159     Value value = mapGet(vm->allModules, moduleName);
160     if (value.type == VT_UNDEFINED) {
161         return NULL;
162     }
163 
164     return VALUE_TO_OBJMODULE(value);
165 }
166 
167 //载入模块 moduleName 并编译
168 static ObjThread* loadModule(VM* vm, Value moduleName, const char* moduleCode) {
169     //确保模块已经载入到 vm->allModules
170     //先查看是否已经导入了该模块,避免重新导入
171     ObjModule* module = getModule(vm, moduleName);
172 
173     //若该模块未加载先将其载入,并继承核心模块中的变量
174     if (module == NULL) {
175         //创建模块并添加到 vm->allModules
176         ObjString* modName = VALUE_TO_OBJSTR(moduleName);
177         ASSERT(modName->value.start[modName->value.length] == '\0', "string.value.
            start is not terminated!");
178 
179         module = newObjModule(vm, modName->value.start);
180         mapSet(vm, vm->allModules, moduleName, OBJ_TO_VALUE(module));
181 
182         //继承核心模块中的变量
183         ObjModule* coreModule = getModule(vm, CORE_MODULE);
184         uint32_t idx = 0;
185         while (idx < coreModule->moduleVarName.count) {
186            defineModuleVar(vm, module,
187                  coreModule->moduleVarName.datas[idx].str,
188                  strlen(coreModule->moduleVarName.datas[idx].str),
189                  coreModule->moduleVarValue.datas[idx]);
190            idx++;
191         }
192     }
193 
194     ObjFn* fn = compileModule(vm, module, moduleCode);
195     ObjClosure* objClosure = newObjClosure(vm, fn);
196     ObjThread* moduleThread = newObjThread(vm, objClosure);
197 
198     return moduleThread;
199 }
200 
201 //执行模块,目前为空,桩函数
202 VMResult executeModule(VM* vm, Value moduleName, const char* moduleCode) {
203     ObjThread* objThread = loadModule(vm, moduleName, moduleCode);
204     return VM_RESULT_ERROR;
205 }
```

…略

仅供非商业用途或交流学习使用

第 158 行定义的 getModule 函数用于从 modules 中获取名为 moduleName 的模块，比较简单，就是把 moduleName 当作 key 从 vm->allModules 中 get 出来，然后将其还原为模块对象。

第 168 行定义的 loadModule 是本节的重点，它用于载入模块 moduleName 并编译。第 171 行通过 getModule 函数试图载入模块 moduleName，这是为了判断模块 moduleName 是否已经被载入了，避免重复导入。如果返回值为 NULL，说明该模块未导入，第 175～191 行执行导入的过程，为 moduleName 创建模块并继承核心模块。第 175～180 行为 moduleName 创建模块 module 并载入到 vm->allModules 中。第 183 行把核心模块导入，目的是在第 184～191 行通过函数 defineModuleVar 把核心模块中的模块变量重新定义到模块 module 中以实现继承。第 194 行调用 compileModule 编译模块，此时还是空的，第 195～198 行表示会单独创建一个线程运行编译的模块。

第 202 行的函数 executeModule 之前是桩函数，这次在其中更新了 loadModule，还不完全，以后再补充。

4.8 虚拟机简介

本章虽然是介绍语法和语义分析，但语义分析中会涉及生成目标指令，进而涉及指令中的操作数，由于是基于栈的虚拟机，因此操作数也会在栈中，为了更好地理解生成指令的过程，必须先介绍虚拟机。

4.8.1 虚拟机分类及优缺点

什么是虚拟机呢？虚拟机顾名思义是虚拟的机器，是软件对硬件的模拟，软件也就是指程序，借助操作系统和编译器提供的功能去模拟硬件的工作，当虚拟机运行时就是个进程，与普通进程无异。由于软件的行为最终也是靠硬件去展现，因此最终虚拟机还是靠硬件完成其功能。

这里所说的虚拟机专指硬件 CPU。为了搞清楚虚拟机做了什么，先要看看硬件 CPU 主要做了什么。执行一个指令大体上有 3 个步骤。

（1）取指。

从程序计数器 PC 指向的地址获取下一条待执行的指令，也就是指令的二进制编码。

（2）译码。

译码即解码，即分析指令的编码，判断指令中的操作码及操作数（指令是由操作码和操作数组成的）。

（3）执行。

指令是一种思想意图，思想要靠行动去落实，“执行”就是 CPU 用行动去展示指令的意义。

注意，在取指阶段中，CPU 是从内存中获取到指令的编码，内存中的是程序的映像，这是由操作系统中的程序加载器负责把程序从文件系统拷贝到内存中的，程序中的是二进制指令流，二进制指令流是被 CPU 原生支持的数据流，也就是说 CPU 知道这种指令流是什么，或者 CPU 按照它自己的理解去处理这个指令流（即使给的是错误的数据），CPU 会自动从这个指令流中获得操作码和操作数，因为这是写入在 CPU 硬件电路中的基因数据，它知道多长的二进制流是操作码，此操作码需要多少个操作数以及操作数在二进制流中的位移和长度，就是说，它懂这个指令流的格式。虚拟机也是循环执行指令的“机器”，那么其指令来自于哪里？格式是什么？这是虚拟机和编译器自己要协调的问题。比如编译器可以以一行一个指令的形式把指令写到文件中，操作码和操作数之间用空格分隔，支持多个操作数，操作数之间也是用空格分隔。虚拟机加载该文件到内存，每次读取一行作为一条指令，以空格去拆分操作码和操作数。看上去不错，至少可行。但是，

仅供非商业用途或交流学习使用

如果操作码的操作数个数不统一的话就涉及指令变长的问题，尽管可以不用理会操作数个数，完全可用一个循环来处理所有的操作数，但这种偷懒是用效率换来的。也许有读者觉得，为了指令的灵活，不如先建立一张表，表中记录操作码和其需要的操作数个数、操作数长度等信息，如果指令有变更，修改这张表就可以了。似乎听上去不错，但实际上这需要虚拟机在执行指令前先通过操作码去索引这张表以确定操作数信息，然后再从指令流中获取操作数，这个查表的过程其实就是“译码”，译码是必然的，每个指令都需要这么复杂的译码过程，因此这对虚拟机速度的影响也是致命的，虚拟机是慢，但不能太过分，恰恰是因为它慢我们才想办法让它不那么慢，因此最好是在虚拟机中写入每一个指令，虚拟机处理到某个操作码后就知道后面有几个操作数。

以上都是在讨论指令流加载到内存后的情况，脚本语言编译器生成的指令流是给虚拟机使用的，按理说编译后马上会执行，那么还需要把指令流输出到文件吗？指令流输出到文件是为了下一次执行脚本时不用再重新编译了，本次用处不大，本次最好是先执行。那么在虚拟机执行后是否把指令流输出到文件以备下次执行直接调用等一系列问题都要自己协调好，任何一个系统都是一套自己协调好的、可以自圆其说的方法集，没有固定的实现方式。我们的方案是，用语义分析生成不定长指令，操作码和操作数不经过文件系统，直接在内存中建立指令流，之后交给虚拟机执行。这么说确实有些抽象，以后实践中读者就明白了。无论怎样，指令流都要加载到内存中了，下面基于此继续讨论。

虚拟机既然要模拟硬件 CPU，那么以下这 3 个步骤也是不可少的，至少在逻辑上要照搬过来。

（1）取指。

从程序计数器 PC（在我们的实现中是用 IP 表示，第 0 章中有介绍）指向的指令流中的地址获得指令。

（2）译码。

判断指令的类型，进入相应的处理流程。

（3）执行。

按照指令的意义执行相应的例程函数。

问题来了，硬件 CPU 执行完指令后，如果指令要输出结果，结果是存放在寄存器或内存中，不过大多数情况下是存储到寄存器中，因为内存 IO 相对于寄存器来说太慢了。虚拟机是软件，说白了就是一个进程，如果执行的指令也要输出结果，结果存储到哪里呢？根据结果存储的地点，虚拟机可分为以下两类。

（1）基于寄存器的虚拟机。

完全按照硬件 CPU 的套路，在虚拟机内部也模拟寄存器，通常是用数组来模拟寄存器组，数组元素便是一个个的寄存器。指令是由操作码和操作数组成的，操作数可以放在模拟的寄存器中，指令执行的结果也可存储到模拟的寄存器中。此类虚拟机代表：lua。

（2）基于栈的虚拟机。

不模拟寄存器，指令的结果存储在所模拟的运行时栈中，操作数可以存储到栈中，也可以直接位于内存中。栈本身是个后进先出的数据结构，执行结果一般会以压栈的形式 push 到栈顶，为维护栈平衡，需要马上被取出，即通过 pop 操作获得结果。值得一提的是，这里所说的“模拟的运行时栈”并不是只有基于栈的虚拟机才有，基于寄存器的虚拟机也要模拟运行时栈，因为这是执行函数或方法所必需的。栈一般用线性结构来模拟，比如数组、链表。此类虚拟机代表：jvm。

这两类虚拟机各有什么优缺点呢？似乎寄存器虚拟机更快，但实际上连虚拟机本身都是在内存中，故以上所模拟的寄存器和栈都是位于内存中的数据结构，从这一层面上说，速度是同一个量级。它们各自的缺点如下。

仅供非商业用途或交流学习使用

（1）基于寄存器的虚拟机实现难度略大。

指令是由操作码和操作数组成的，两类虚拟机的操作数分别用不同的结构来存储，因此这涉及指令生成的难度不同。基于寄存器的虚拟机所模拟的寄存器和真实系统一样，因为寄存器的个数是有限的，要把无限的变量分配到有限的寄存器中，如何避免不冲突，这涉及寄存器分配的算法，比如经典的图着色算法，因此难度会大一些。

（2）基于栈的虚拟机性能略低。

首先栈是由内存结构模拟的，压栈和弹栈都要分两步，先移动栈顶指针再获取数据，传递数据时需要把操作数压栈，然后再由接收方弹栈，因此多了一次操作。

其次操作数放在寄存器中还有优化的余地，放在栈中的话位置就不能动了，一般是栈顶和次栈顶（位于栈顶之上，仅次于栈顶的已用 slot），因此整体上优化潜力略低。

4.8.2　为什么要采用虚拟机

我们知道，虚拟机通常是为那些脚本语言准备的，比如 PHP、Perl、Python 等。对于编译型语言就不需要虚拟机了，比如 C 和 C++是直接在硬件 CPU 上运行的。

有人说“软件肯定比硬件慢”，我学得这句话不妥。软件是用来描述人的思想，思想是靠硬件转变成“行为”的。软件的意图是靠硬件来表达，软件是驾驭硬件、指挥硬件运行的思想，换句话说，如果硬件做出的行为很慢，那肯定是软件的思想就不快。虚拟机也是软件，但为什么人们常说虚拟机比真实的物理机要慢呢？虚拟机比物理机慢，是因为虚拟机这个软件服务的对象不同，也就是角色不同，它的客户从没发现它是虚拟机，一直就把它当作真实的物理机来对待。在细节上虚拟机为什么慢？程序最终会被编译为指令，几乎每条指令中的操作数都会涉及寄存器，在虚拟机中模拟的寄存器是某种数据结构（比如数组，但形式不限），该数据结构必然是在内存当中，内存操作比真实的寄存器操作要慢上几个数量级，CPU 更愿意用寄存器是因为寄存器位于 CPU 内部，和 CPU 速度是一致的，一个操作仅需要一个时钟周期。而内存是在外部，而且内存操作要通过总线的方式，等上几十个时钟周期后内存才收到读写的消息，最要命的是，内存读写过程中要花费掉的时间对 CPU 来说是天文数字，等拿到数据时 CPU 都老了。在虚拟机中，为了维护好所模拟的寄存器，必然要额外写一些相关的维护指令，比如为寄存器计数，以此来判断该寄存器的使用频率，这可作为下次分配寄存器时的判断依据。还有更要命的是，如果所有寄存器都被使用了，要考虑先置换出一个，先腾出空间来供本次指令的执行等等，这都要一系列算法，算法本身位于解释器中（也许是编译器中，也许是虚拟机中），其指令数细化到物理 CPU 上是指数级的，这对单纯的寄存器读写操作来说，产生了大量不可忍受的冗余，因此虚拟机寄存器一个读写涉及更多更慢的内存操作，而物理机寄存器操作仅是直接一步，必然要比虚拟机快太多。不可避免的是几乎每一条指令都包含寄存器，虚拟机执行每条指令都有这样的“拖沓”，这对于十几万条指令的程序来说，速度上的差别是显而易见的。而且，如果一些处理算法是在虚拟机中被调用，也就是在运行时阶段“随机应变、临场发挥”，这比起在编译阶段就搞定这些问题的二进制可执行程序来说，更加降低了效率。实际工作中，为了稳定，我们经常要做一些预案来应付不测，但实际上没有人期望预案真被执行，预案只是最大化地止损措施，这多少都会拖慢工作计划的进行。运行时调用的一些方法其实就是一些提前做好的预案，能少调用就少调用，不调用最好，就像“出门前规划好一条很少出意外的路线”往往要比“在必然充满意外的捷径上不断解决问题”能更快到达目的地。

那么，为什么还要采用虚拟机呢？原因如下。

（1）可移植性

程序开发中经常要用到软件包，通常我们是在某一平台上开发程序，在该平台上的软件包并不表示在另一平台上也有，比如 Linux、Windows 和 Mac 作为主要的开发平台，同一种语言也不

仅供非商业用途或交流学习使用

完全兼容对方的软件包，都有自己的非标准扩展。运行程序的平台如果不是开发的平台，那么就需要重新编译，如果涉及不同平台的软件包，代码需要从一个平台移植到目标平台，“移植”其实就是按照目标平台重写代码，这对于十万行以上的工程就很吃力了，对于百万至千万行工程来说是难以想象的。需要移植的根本原因是在各个平台上的目标指令不一致，可是使不同平台的目标指令都一样是不可能的，否则就是同一个平台了，因此一种解决办法就是加一个中间层，让所有平台都支持这个中间层，开发人员只写出适合此中间层的代码即可，那么就不需要花费大量精力维护不同平台间的可用性了，只要维护好这个中间层即可，完全做到“一次编译，到处运行”。虚拟机就是这个中间层，适合此中间层的代码就是脚本语言和操作码，大概过程是，由虚拟机的研发人员在各个平台上开发出逻辑相同的虚拟机，业务开发人员使用脚本语言开发产品，由脚本语言编译器把脚本语言编译为操作码（也称 opcode），最终由虚拟机来执行操作码。此操作码可以放在不同平台的虚拟机上直接运行。

（2）易于开发出功能强大的脚本程序

脚本语言比编译型语言的语法更简洁，能用更短的代码实现较多的功能，对于开发人员来说更省事儿，使用起来更方便。这背后的原因如下。

① 虚拟机多数是用高级语言开发的，高级语言的功能较强大，而且还有很多现成的软件包可用。比如脚本语言中“hello，”+“world。”就可以把两个字符串连接成“hello，world。”，脚本语言要实现这样的功能，在其背后的高级语言（如 C 语言）可以调用 strcat 函数来直接实现两个字符串的连接，要是用底层汇编的话就麻烦多了，因此用高级语言编写的解析器（包括词法分析器、语法及语义分析器、虚拟机）容易构建出功能强大的脚本语言。

② 虚拟机只是执行操作码，一个操作码在虚拟机层面往往是由多个函数来实现的，这意味着一个操作码的功能是非常强大的，而操作码是由脚本语言生成的，因此脚本语言的功能非常强大也是必然的。

③ CPU 执行指令的 3 个步骤是取指、译码和执行。其中前两个步骤是非常耗时的。在虚拟机中也会有这 3 个步骤，同样这 3 个步骤中的取指和译码是虚拟机中最耗时的部分，不过与 CPU 不同的是，虚拟机可以规避这些耗时的部分，它不提供这样的指令，由前端语法和语义分析直接处理，不需要经过虚拟机，比如对于表达式 1+2 来说，CPU 要用指令 add 来做，这涉及“取指，译码，执行”，然而虚拟机可以没有 add 操作码（当然也可以有，取决于实现），可以在语义分析阶段直接生成开发语言的表达式“1+2”，避免了“取指，译码”，相当于直接“执行”了。

（3）充分利用硬件资源

这个优势主要体现在模拟操作系统的虚拟机中，前两种优势主要体现在语言解析器方面。虚拟机若根据执行的指令类型来划分，一种是执行 CPU 的真实指令，这种虚拟机用来模拟操作系统，比如 vmware 或 virtualBox 等。另一种就是执行自定义的操作码，就是本书所说的语言解析器。由于一个进程在任意时刻只占用一个 CPU 核心，而且进程很难被拆分成多个子任务分配在多个核心上并行工作，因此多核 CPU 中大量的核心都是空闲的，可以通过多开几个虚拟机实例的方式充分利用 CPU 等硬件资源，实现一机变多机以节约成本。

（4）大部分情况下，虚拟机并不是整个系统的瓶颈

虚拟机是为脚本语言服务的，尽管在执行脚本语言时显现了速度上的不足，但这并不重要，虚拟机执行脚本语言毕竟是属于内存中的操作，有比它还慢的，而且还慢得很多，比如系统中免不了的磁盘操作和网络传输，脚本语言在执行这类操作时都要被操作系统阻塞（确切地说，是虚拟机被操作系统阻塞，操作系统只能看到虚拟机进程），以等待这些低速设备工作的完成，因此脚本语言还是相对很快的。

综上，虚拟机存在即合理。

仅供非商业用途或交流学习使用

4.8.3　虚拟机的简单优化

出于开发简单的考虑，我们就选择基于栈的虚拟机，效率可以在其他地方弥补。

指令是由操作码和操作数组成的，无论在虚拟机还是在硬件 CPU 中，操作数的个数和种类都是由操作码决定的，因此操作码才是虚拟机和 CPU 译码的关键。虚拟机的 3 个步骤是取指、译码和执行，前两个步骤是最耗时的，操作码种类很多的话将是不小的开销。如果在设计阶段就把操作码种类减少，也就是想办法让每个操作码做更多的工作，这样总体上脚本程序编译后的指令数就会大大减少，操作码少了也就减少了花费在取指、译码上的时间，因此提升了效率。另外，操作码种类减少后，即使整体上脚本程序编译后的指令数量变化不大，但却能提高指令的重复率，即同一个指令被执行的次数更多了，这大大提高了程序的局部性。局部性原理是指最近被执行到的往往在一段时间内也被持续执行到，程序计数器 PC 指向了程序中这块被反复执行的“局部”。比如在循环结构中的指令是一段时间内被反复执行的指令，这些指令就是局部。局部性的特点是“反复”，这说明具备了“可预测”的特性，可预测的特性经常被用在缓存和分支预测当中。由于可预测，因此先缓存起来以备下次重复访问，CPU 的缓存机制也是利用程序的局部性原理，因此减少了操作码的种类，有利于提高 CPU 的缓存命中率。另一个就是分支预测了，现在 CPU 都是多级流水线并行“取指、译码、执行”，也就是这 3 个步骤是重叠执行的，因此流水线级数体现了 CPU 同时执行指令条数的能力，也就是 CPU 的吞吐量，一级流水线执行一条指令。流水线主要是为 CPU 中的分支预测服务的，根据预测结果 CPU 在执行当前指令的同时预先把下一条指令的译码和下下条指令的取指放在流水线上，如图 4-6 所示。

指令	周期1	周期2	周期3	周期4	周期5
第一条指令	取指	译码	执行		
第二条指令		取指	译码	执行	
第三条指令			取指	译码	执行

图 4-6

如果 CPU 预测失败，就要把已经加载到流水线上的操作清空。尽管这里说是 3 个步骤，但这只是宏观上的，CPU 为了挖掘性能已经把这 3 个步骤细化到更多级的微步骤，这和 CPU 厂商的工艺有关，有 8 级、16 级、20 级甚至 31 级，也就是说大部分 CPU 中流水线级数已经不是 3 级，级数越多，指令并行能力越强。在预测成功的情况下会大大提高速度，但凡预测失败就要把已经上了流水线的微操作清空，代价很大。程序局部性的提高有利于分支预测的成功率，减少了清空流水线的成本，这就相当于间接为程序提速。缓存和分支预测都是利用了程序局部性原理，局部性只是暂时的，最后总会失败，总有一次预测是白费力气，也就是清空流水线，但毕竟大多数情况下是预测成功的，比如一个 while 循环要循环执行 10 次循环体，尽管 CPU 也把预测失败的“第 11 次循环”执行的循环体指令放在流水线上了，但前 9 次（一般是第 2 次循环时 CPU 才大胆预测局部性）都成功了，利远大于弊。另外读者想想看，程序中具有局部性的代码太多了，因为程序主要是由“循环、选择、顺序”3 种结构组成的，循环结构肯定具有局部性，因此具有局部性的代码要占 1/3 左右。局部性会从这里移到那里，总是能发挥流水线的价值。

在虚拟机的译码阶段还可以优化，在执行完指令后可以通过 goto 无条件跳转到译码的入口，这避免了先跳到译码结束，再回到译码入口的二次跳转。

仅供非商业用途或交流学习使用

我们选择的是基于栈的虚拟机。

4.9 字节码

字节码也称 byte code，是一种虚拟机语言中经常采用的中间代码，比如 PHP、Python 等。字节码是语义分析的输出，同时也是虚拟机的输入，也就是说语义分析的目标代码就是字节码。字节码只绑定于特定的虚拟机，即仅适用于特定虚拟机，并不绑定到特定的 CPU 硬件，因此字节码是自定义的，形式不限。指令是由操作码和操作数组成的，操作码称为 opcode，操作数称为 operand，前面说过了，为提升虚拟机的效率，操作码种类尽量要少，基本上都小于 256 个，因此操作码用 1 个字节便能表示了，这也正是字节码中“字节”二字的缘由。

我们也要定义自己的字节码，在定义之前有一些说明：这是一个基于栈的虚拟机，操作码的部分操作数、运算结果都会保存在栈中。运行时栈是由线性结构（一般为数组或链表）模拟的，线性结构中的单位（元素或节点）称为 slot，我们这里是用数组来模拟栈。

指令对栈空间有一定的影响，有的操作码会往栈中 push 数据，因此使栈空间增大一些 slot，有的操作码会从栈中 pop 数据，因此会减小一些 slot。函数是由一系列的指令组成的，函数中的局部变量存储到函数的运行时栈中，函数所需要的运行时栈空间大小除了要考虑局部变量的个数外，肯定也要考虑操作码对栈的影响，因此要列出操作码对栈的影响。操作码的行为是一定的，因此各操作码对栈的影响也是一定的。

符号表是个笼统的概念，任何存储符号信息的结构均可称为符号表。很多操作码的操作数是各符号表中的索引 index，由虚拟机在执行阶段通过 index 到相应符号表中获得数据。为陈述方便：

（1）索引为 index 的栈元素用 stack[index]表示。

（2）索引为 index 的常量用 constants[index] 表示。

（3）索引为 index 的模块变量的值用 moduleVarValue[index]表示。

提前说明一下，我们对模块变量的管理分为名字和值两部分，模块变量名是放在符号表 moduleVarName 中的。

（4）索引为 index 的方法名用 allMethodNames[index]表示。

（5）索引为 index 的域用 fields[index]表示。

（6）索引为 index 的 upvalue 用 upvalues[index]表示。

（7）索引为 index 的方法用 methods[index]表示。

（8）索引为 index 的指令用 instrStream[index]表示。

举个例子，当虚拟机执行指令 LOAD_CONSTANT index 时，虚拟机分别获得了操作码 LOAD_CONSTANT 和操作数 index，然后以 index 作为索引去常量表 constants 中检索，获得待加载到栈中的常量 constants[index]，然后将 constants[index]压栈。

我们所有的字节码都定义在表 4-1 所示的字节码表中了，先说明下这个表结构，每个指令独立成一个单元，每 3 行是一个指令单元，各指令由空行分隔。指令单元内容包括 3 部分，即 3 行。

第一行是指令格式：“操作码 操作数”。我们要实现基于栈的虚拟机，因此操作数包括位于指令流中的操作数和位于栈中的操作数。

第二行是指令的意义。

第 3 行是指令对运行时栈大小的影响，比如该指令有 N 个压栈动作就称使栈增大 N 个 slot，对栈有 N 个弹栈动作就称使栈减少 N 个 slot。

仅供非商业用途或交流学习使用

表 4-1　　字节码表

操作码	操作数（操作码无相关操作数则为空白）	
	指令流	栈
LOAD_CONSTANT		
	把索引为 index 的常量（即 constants[index]）压栈	
	只能加载一个操作数入栈，故使栈增大 1 个 slot	
PUSH_NULL		
	往栈中压入 null	
	压栈使栈增大 1 个 slot	
PUSH_FALSE		
	往栈中压入 false	
	压栈使栈增大 1 个 slot	
PUSH_TRUE		
	往栈中压入 true	
	压栈使栈增大 1 个 slot	
LOAD_LOCAL_VAR	index	
	把索引为 index 的局部变量的值（即 stack[index]）压栈	
	压栈使栈增大 1 个 slot	
STORE_LOCAL_VAR	index	栈顶
	把栈顶数据存入索引为 index 的局部变量（即 stack[index]）中	
	此指令并不会压栈或弹栈，故不改变栈大小	
LOAD_UPVALUE	index	
	把索引为 index 的 upvalue（即 upvalues[index]）压入栈	
	压栈使栈增大 1 个 slot	
STORE_UPVALUE	index	栈顶
	把栈顶数据存储到索引为 index 的 upvalue（即 upvalues[index]）中	
	此指令并不会压栈或弹栈，故不改变栈大小	
LOAD_MODULE_VAR	index	
	把索引为 index 的模块变量的值（即 moduleVarValue[index]）压栈	
	压栈使栈增大 1 个 slot	
STORE_MODULE_VAR	index	栈顶
	把栈顶数据存入到索引为 index 的模块变量（即 moduleVarValue[index]）中	
	此指令并不会压栈或弹栈，故不改变栈大小	

续表

操作码	操作数（操作码无相关操作数则为空白）	
	指令流	栈
LOAD_THIS_FIELD	index	
	把 this 对象中索引为 index 的域（即 this.fields[index]）压栈	
	压栈使栈大小增加 1	
STORE_THIS_FIELD	index	栈顶
	把栈顶数据存储到 this 对象中索引为 index 的域（即 this.fields[index]）中	
	此指令并不会压栈或弹栈，故不改变栈大小	
LOAD_FIELD	index	
	弹出栈顶元素作为实例对象，并将该实例对象中索引为 index 的域（即该对象.fields[index]）压栈	
	一弹一压，栈大小不变	
STORE_FIELD	index	栈顶，次栈顶
	弹出栈顶元素作为实例对象，并且把次栈顶的数据存入该实例对象中索引为 index 的 field（即该对象.fields[index]）中	
	弹出了栈顶数据，故使栈大小减 1	
CALL0	index	栈顶～以下是所调用方法的实参数组 args，占用的栈 slot 数量以参数个数+1 为准，+1 原因是 this 对象
CALL1		
CALL2		
CALL3		
CALL4		
CALL5		
CALL6		
CALL7		
CALL8		
CALL9		
CALL10		
CALL11		
CALL12		
CALL13		
CALL14		
CALL15		
CALL16		
针对以上 17 个 CALL 操作码	调用本类中索引为 index 的方法（即 this.methods [index]），方法执行后，主调方会弹出参数所占的栈空间	
	CALL 族操作码形如“CALL 参数个数”，注意，“参数个数”是 CALL 所调用的方法所需要的参数个数（并不是 CALL 本身需要的操作数个数。CALL 族需要的是一个 2 字节大小的操作数），这些参数位于栈中。执行完成后会弹出所有参数，故使栈减少“参数个数”个 slot	

续表

<table>
<tr><td rowspan="2">操作码</td><td colspan="3">操作数（操作码无相关操作数则为空白）</td></tr>
<tr><td colspan="2">指令流</td><td>栈</td></tr>
<tr><td>SUPER0</td><td rowspan="16">method
Index</td><td rowspan="16">super
Index</td><td rowspan="16">栈顶以下是所调用方法的实参数组 args，占用的栈 slot 数量以参数个数+1 为准，+1 的原因是 this 对象</td></tr>
<tr><td>SUPER1</td></tr>
<tr><td>SUPER2</td></tr>
<tr><td>SUPER3</td></tr>
<tr><td>SUPER4</td></tr>
<tr><td>SUPER5</td></tr>
<tr><td>SUPER6</td></tr>
<tr><td>SUPER7</td></tr>
<tr><td>SUPER8</td></tr>
<tr><td>SUPER9</td></tr>
<tr><td>SUPER10</td></tr>
<tr><td>SUPER11</td></tr>
<tr><td>SUPER12</td></tr>
<tr><td>SUPER13</td></tr>
<tr><td>SUPER14</td></tr>
<tr><td>SUPER15</td></tr>
<tr><td>SUPER16</td></tr>
<tr><td>针对以上 17 个 SUPER 操作码</td><td colspan="3">把索引为 superIndex 的常量（即 constants[superIndex]）作为基类 super，调用 super 中索引为 methodIndex 的方法（即 super.methods[methodIndex]），执行后由主调方回收参数所占的栈空间</td></tr>
<tr><td colspan="3">操作码形如“SUPER 参数个数”。“参数个数”的意义同 CALL 族操作码一样。执行完成后会弹出所有参数，故使栈减少“参数个数”个 slot</td></tr>
<tr><td rowspan="3">JUMP</td><td>offset</td><td colspan="2"></td></tr>
<tr><td colspan="3">无条件向前跳转：
offset 是跳转“正”偏移量，即向前（往高地址）跳到指令流中 ip += offset 处（即 &instrStream[ip+offset]），其中 ip 是下一条指令地址</td></tr>
<tr><td colspan="3">跳转指令不改变栈大小</td></tr>
<tr><td rowspan="3">LOOP</td><td>offset</td><td colspan="2"></td></tr>
<tr><td colspan="3">无条件向回跳转：
offset 是跳转“正”偏移量，即向回（往低地址）跳到指令流中 ip-= offset（即& instrStream [ip-offset]）处</td></tr>
<tr><td colspan="3">跳转指令不改变栈大小</td></tr>
<tr><td rowspan="3">JUMP_IF_FALSE</td><td>offset</td><td colspan="2">栈顶</td></tr>
<tr><td colspan="3">offset 是跳转“正”偏移量，将栈顶弹出作为条件，若条件为 false 则向前（往高地址）跳到指令流中 ip+=offset（即& instrStream [ip+offset]）处</td></tr>
<tr><td colspan="3">弹出了栈顶数据，故使栈大小减 1</td></tr>
<tr><td rowspan="3">AND</td><td>offset</td><td colspan="2">栈顶</td></tr>
<tr><td colspan="3">这是“与运算”的短路方式：左操作数为 false 时右操作数就不用测试了。offset 是跳转“正”偏移量，如果栈顶处的值（即“&&”的左操作数）是 false，向前（往高地址）跳到指令流中 ip+=offset（即&instrStream[index]）处，即跨过右操作数的计算步骤，否则弹栈并继续计算右操作数</td></tr>
<tr><td colspan="3">若弹栈则使栈大小减 1</td></tr>
</table>

续表

操作码	操作数（操作码无相关操作数则为空白）	
	指令流	栈
OR	offset	栈顶
	这是“或操作”的短路方式：左操作数为 true 时，右操作数就不用测试了。如果栈顶处的值（即||的左操作数）是 true，向前（往高地址）跳到指令流中 ip+=offset（即&instrStream[index]）处，即跨过右操作数的计算步骤，否则弹栈并继续计算右操作数	
	若弹栈则使栈大小减 1	
CLOSE_UPVALUE		栈顶
	栈顶中存储的是地址界限，关闭地址大于地址界限的 upvalue，然后弹栈	
	弹栈使栈大小减 1	
RETURN		栈顶
	退出当前函数并弹出栈顶的值作为返回值	
	RETURN 操作码会弹出栈顶的值作为返回值，但是其对栈的影响并不是减少一个 slot，而是 0。原因是 RETURN 用于从函数返回，无论是回到主调方，还是结束程序运行即从程序的入口（相当于主函数）中返回，都是结束函数调用。我们这里统计操作码对栈的影响，目的是在函数调用前，给函数分配运行时栈所需的空间大小做参考。说明一下，主调方函数和被调方函数的运行时栈在空间上是接壤的，被调方函数的返回值是存储到被调方函数自己的运行时栈的栈底，所以此处恰是主调方函数运行时栈的栈顶，这样主调方函数就可以到自己的栈顶获得被调方函数的返回值。操作码 RETURN 的使命除了要返回栈顶的值做返回值外，还要回收所属函数的运行时栈。为了保证能把返回值传给主调方，RETURN 回收的是除用于保留返回值的栈底 slot 外的整个运行时栈，因此对栈的影响取决于实际的栈大小，即回收的 slot 数量不定。但读者要明白，搞清楚 RETURN 对栈的精确影响并不重要，重点是我们统计操作码对栈的影响目的是确保运行时栈在函数“运行的时候”有足够的空间，而 RETURN 是函数中最后一个指令了，函数不在“运行的时候”了或者是函数处于“运行时”的尾声，因此需要 RETURN 做善后工作，也就是刚才所说的回收运行时栈的空间，而不是再去占用运行时栈的空间，因此没必要再统计。因为函数最终的指令流也是由这些操作码组合的，我们为统计函数所需运行时栈的大小，对所有操作码都要统计对运行时栈的影响，为不影响统计结果，就强制其为 0	

CREATE_CLOSURE	closureIndex，
	{
	[bool1，upvalueIndex1]，
	[bool2，upvalueIndex2]，
	[bool3，upvalueIndex3]，
	…
	}	

	把索引为 index 的常量（常量 constants[index]中的函数是待创建闭包的函数）创建闭包，然后将创建的闭包压栈。上边大括号中的内容可选的，其中的中括号对儿是布尔值和 upvalue 索引。每一对中括号表示一个 upvalue 的信息，如果布尔值为 true，upvalueIndex 为直接外层函数中局部变量的索引，否则为直接外层函数的 upvalue 的索引。中括号对儿的数量取决于待创建闭包函数中的 upvalue 数目，即 constants[index].upvalueNum，如果 upvalueNum 为 0 则没使用到 upvalue，所以说大括号可选	
	压栈操作使栈增大 1	

续表

操作码	操作数（操作码无相关操作数则为空白）	
	指令流	栈
CONSTRUCT		栈底
	类对象在当前运行时栈的栈底（即 stack[0]），该操作码创建一个类实例，然后用该类实例去替换栈底的类对象	
	只是原地替换 stack[0]，不改变栈大小	
CREATE_CLASS	域（字段）数量	栈顶、次栈顶
	创建一个类，栈顶是基类，次栈顶（距离栈顶最近的已用的 slot）是类名，指令流中的操作数指定了域的个数。基类所在的栈顶 slot 会被回收，创建的类存储到原来次栈顶所在的 slot	
	此操作码会回收基类所占的栈顶空间，因此使栈大小减 1	
INSTANCE_METHOD	index	栈顶、次栈顶
	函数与方法的区别是方法有所属的类，所以创建方法就是把方法绑定到某个类。为索引为 index 的方法名（即 allMethodNames [index]）创建方法，栈顶是待绑定的类，次栈顶是待绑定的方法，完成后弹出栈顶和次栈顶	
	两个弹栈操作使栈减少两个 slot	
STATIC_METHOD	Index	栈顶、次栈顶
	同操作码 INSTANCE_METHOD 类似，都是为方法名 allMethodNames [index]创建方法，栈顶是所绑定的类，次栈顶是所绑定的方法，完成后弹出栈顶和次栈顶。与 INSTANCE_METHO 的差异是，STATIC_METHOD 将方法所绑定的类指向该类的 meta 类	
	两个弹栈操作使栈减少两个 slot	
END		
	伪指令，表示代码结束	
	伪指令不会被执行，故不改变栈大小	

第5章 自上而下算符优先——TDOP

5.1 自上而下算符优先——TDOP

自上而下算符优先即 TopDownOperatorPrecedence，编写 TDOP。它是一种语法分析技术，可用来构造一棵语法树，并不负责编译生成中间代码。从名字上看是结合了两种语法分析方法：自上而下和算符优先。注意，算符优先分析法采取的是自下而上的归约方式，TDOP 并未采纳这个方式，而是汲取了其优先级的特性。因此 TDOP 的特点是自上而下的推导以及优先级比较。尽管一提到优先级就让人想起数学中的运算符，但语法分析中的优先级并不仅限于数学运算符，在 TDOP 中有很多语法关键字也具有优先级。

5.2 来自 Douglas Crockford 的教程

以下内容用 JavaScript 语言采用 TDOP 技术实现的 JavaScript 语法分析器。本小节是 Douglas Crockford 于 2007 年 2 月 21 日写的，Douglas Crockford 是 Web 开发领域知名的技术权威，ECMA JavaScript2.0 标准化委员会委员。被 JavaScript 之父 Brendan Eich 称为 JavaScript 的大宗师（Yoda）。曾任 Yahoo!资深 JavaScript 架构师，现任 PayPal 高级 JavaScript 架构师。

本小节从实践的角度诠释了 TDOP 的技术，样例简单经典，作为 TDOP 教程非常合适。以下是我翻译过来的。

实话实说，TDOP 还是有一定难度的，因此我在原文中追加了一些内容帮助理解，为了叙述样例代码的逻辑，有的源码中我加了行号，如果逻辑比较清楚或者不重要的代码就没有行号，大家可以对照原版辩证地看。此教程只用于帮助理解 TDOP 原理，大家不要过度消耗精力在代码语法上，咱们有自己的实现方式，而且原文并未给出全部的代码，全部的代码文件有两个，分别是 parse.js 和 tokens.js。

有兴趣的读者自己看看吧，提前说下，对我们来说本教程的重点有：

（1）expression 函数；

（2）中缀运算符和前缀运算符中的 nud 方法和 led 方法；

（3）字面量的处理方法。

理解 TDOP 的精髓是有些难度的，如果阅读本小节后没完全搞清楚也没关系，本节就是让大家先从整体上认识 TDOP，下一节我会用自己的方式帮助大家梳理重点。下面是原文翻译。

1973 年于波士顿，Vaughan Pratt 在编程语言原则座谈会第一期年刊上发表了论文“自上而下算符优先”（简称 TDOP），这是一种结合了“递归下降”和“floyd 算符优先”优点的语法解析技术。它便于使用，看上去像递归下降语法解析，但是用了更少的代码实现了更高的性能。他声称这项语法解析技术简单易懂，易于实现，使用简单，极其高效且灵活，它是动态的，支持语言的

真正扩展。

奇怪的是，对编译器构造来说，这么高效的方法在今天居然被忽略了，什么原因？Pratt 在论文中指出，由于 BNF 及其变种语法先入为主，大家普遍采用 BNF，随着自动机理论的出现，大家热衷于自动机，这导致了自动机之外的技术未得到发展。另一种解释说，他的技术对动态函数式语言更为有效，对于静态过程式语言将非常困难。

直到 JavaScript 的出现这种情况才改变。JavaScript 是一门动态函数式语言，其语法属于 C 语言语法家族，编程社区都很喜欢此语法。Pratt 在 1973 年预测面向对象是未来的趋势，JavaScript 拥有面向对象的特性，是一种可以展示 TDOP 技术的理想语言，下面就用 JavaScript 实现一个简单的语法分析器。

符号表

每种 token，比如运算符和标识符，都从一个符号对象继承，我们在符号表中记录所有符号对象，首先要定义符号表 symbol_table。

```
1 var symbol_table = {};
```

symbol_table 类型为{}，即哈希数组，用来存放下面的 original_symbol 对象及其子类，符号表的 key 为“字符串”，value 为 original_symbol 对象及其子类。

```
1 var original_symbol = {
2     nud: function () {
3         this.error("Undefined.");
4     },
5     led: function (left) {
6         this.error("Missing operator.");
7     }
8};
```

上面是以对象直接量的形式定义 original_symbol，即生成的是对象，从名字上可知，original_symbol 是原始的符号，符号都是基于它的扩展，original_symbol 相当于“符号”的“基类”。这种 key:value 形式的对象直接量语法在 javaScript 中用得很广泛，function 表示定义一个函数，函数在此相当于 value，绑定到 key，key 是 nud 和 led，即在 original_symbol 类中分别定义了 nud 和 led 方法。nud 方法表示不关注左操作数，led 方法表示关注左操作数，后面会细说这两个方法。

接下来定义函数 symbol，用来生成符号对象并向符号表中添加符号对象。

```
01 var symbol = function (id, bp) {
02     var s = symbol_table[id];
03     bp = bp || 0;
04     if (s) {
05         if (bp >= s.lbp) {
06             s.lbp = bp;
07         }
08     } else {
09         s = Object.create(original_symbol);
10         s.id = s.value = id;
11         s.lbp = bp;
12         symbol_table[id] = s;
13     }
14     return s;
15};
```

symbol 包含参数 id 及 bp，id 表示符号（identity），是个字符串。bp 表示绑定权值（bind power），绑定权值“简单理解”就是运算符吸引操作数的能力，也就是运算符的优先级，这里所说的简单理解只是为了帮助理解绑定权值的概念，并不是只有运算符（特指数学运算符）才有操作对象（操作数），涉及操作对象的符号都有绑定权值，比如 class.method 中的“.”就有非常高的绑定权值，这是后话，咱们继续。参数 bp 是可选的，可见第 3 行默认为 0。第 4 行如果符号表中字符串 id 对应的符号已存在，也就是曾经已将同字符串封装为符号 s 了，就返回 id 对应的符号对象，否则第 9～12 行创建 id 的符号对象并加入到符号表 symbol_table，随后返回新创建的对象。符号 s 是用 Object.create 创建的对象，可以动态添加属性（域），第 10～11 行为符号添加了 id、value 和 lbp 属性。对于关注左操作数的运算符，比如中辍运算符，通常情况下，绑定权值 bp 要赋给符号的左绑定权值 lbp。不过在本教程中符号没有 rbp 即右绑定权值属性，使用时是直接用 bp 代替的。

下面往符号表中添加一些符号。

```
symbol(":");
symbol(";");
symbol(",");
symbol(")");
symbol("]");
symbol("}");
symbol("else");
```

下面（end）表示文件结束，再没有多余符号，（name）是新标识符的原型，比如变量名。其中的括号是为避免和用户源码冲突。

```
symbol("(end)");
symbol("(name)");
```

Tokens

我们假设已经经过词法分析，源代码已经被转换成 token 存储到 tokens 数组。每个 token 包含一个表示对象类型的成员 type 和表示对象值的成员 value。type 可以是标识符（即名称）、字符串、数字或运算符，value 可以是字符串或数字。

语法分析就是把 token 转换成语法树。我们需要遍历源码中所有的 token，为此我们要定义一个全局变量 token 表示当前的 token。

```
var token;
```

下面还需要定义一个移进 token 的方法（就像 sparrow 中 getNextToken），在此它是 advance。

```
01 var advance = function (id) {
02     var a, o, t, v;
03     if (id && token.id !== id) {
04         token.error("Expected '" + id + "'.");
05     }
06     if (token_nr >= tokens.length) {
07         token = symbol_table["(end)"];
08         return;
09     }
10     t = tokens[token_nr];
11     token_nr += 1;
12     v = t.value;
```

```
13      a = t.type;
14      if (a === "name") {
15          o = scope.find(v);
16      } else if (a === "operator") {
17          o = symbol_table[v];
18          if (!o) {
19              t.error("Unknown operator.");
20          }
21      } else if (a === "string" || a ===  "number") {
22          a = "literal";
23          o = symbol_table["(literal)"];
24      } else {
25          t.error("Unexpected token.");
26      }
27      token = Object.create(o);
28      token.value = v;
29      token.arity = a;
30      return token;
31  };
```

advance 可以更新 token 的值，同时为它添加 arity 信息。arity 表示运算元，这个元是指 token 的元信息，以此来说明 token 的类型、操作数个数等等，通过它可以明确该 token 的类型及用途。此方法还可以接受一个参数 id，表示期望的下一个 token（类似 sparrow 中的 consumeCurToken）。第 02 行的 a 表示 arity，o 表示 original_symbol，v 表示 value，t 表示 token。前面说过，此教程是假设所有 token 已经存储到了数组 tokens 中（因此不用纠结 token 是如何识别出来的，有兴趣可参阅本章开头所说的 tokens.js）。token 是用下标 token_nr 访问的，token_nr 表示下一个 token 的索引。第 10 行从数组 tokens 中获得下一个 token 到 t，同时在第 11 行更新 token_nr 使其自加 1。注意，尽管 t 是最新的 token，它并没有被直接返回，而是利用其属性创建了一个新的 token，最终返回的是新 token。在第 12～13 行分别获得了其 value 和 type 到 v 和 a 即 arity 中，随后在第 14～23 行根据其类型创建不同的符号 o，最后第 27～29 行将符号 o 封装为对象后，用 v 和 a 分别为对象的 value 和 arity 赋值，最后返回 token。

作用域

为定义符号，比如变量名，大部分语言都有一些描述方法，对于简单的语言，当我们遇到一个新词时，就定义它并添加到符号表中。在复杂的语言中，我们希望有作用域让程序员方便控制变量的生存期及可见性。

作用域就是变量在程序中定义及可访问的区域，作用域支持嵌套，定义在内层作用域的变量对外层不可见。

在此先定义函数 itselt，很多方法都用到它，它返回对象自己。

```
var itself = function () {
    return this;
};
```

我们将当前作用域保存在变量 scope 中，定义全局变量 scope。

```
var scope;
```

下面的 original_scope 是所有作用域对象的原型，它包括很多方法，下面把方法分拆解释。

define 方法用于在作用域中定义变量，将标识符 token 转换为变量 token，如果变量重定义或者是系统保留字就报错。

```
var original_scope = {
     define: function (n) {
          var t = this.def[n.value];
          if (typeof t === "object") {
              n.error(t.reserved ?
                   "Already reserved." :
                   "Already defined.");
          }
          this.def[n.value] = n;
          n.reserved = false;
          n.nud      = itself;
          n.led      = null;
          n.std      = null;
          n.lbp      = 0;
          n.scope    = scope;
          return n;
     },
```

find 方法用于查询符号 name 的定义，从当前作用域开始查找，如有必要，会一直顺着父作用域链向上，直到符号表的最外层。如果未找到则返回 symbol_table["(name)"]。

同时 find 方法还会测试变量，确认其不是未定义（可能是未声明的符号），也不是函数（和继承的方法冲突）。

```
    find: function (n) {
            var e = this, o;
            while (true) {
                o = e.def[n];
                if (o && typeof o !== 'function') {
                      return e.def[n];
                }
                e = e.parent;
                if (!e) {
                     o = symbol_table[n];
                     return o && typeof o !== 'function' ?
                              o : symbol_table["(name)"];
                }
            }
        },
```

pop 方法关闭当前作用域，回到父作用域。

```
    pop: function () {
         scope = this.parent;
    },
```

reserve 方法用于表示一个符号在当前作用域中被用于保留字。符号变成保留字的原理就是把该符号提前定义，由于语法规定不可重复定义相同的符号，因此用户就定义不了该符号了，从而实现了保留字不可用的效果。

```
    reserve: function (n) {
        if (n.arity !== "name" || n.reserved) {
            return;
        }
        var t = this.def[n.value];
        if (t) {
            if (t.reserved) {
                return;
            }
            if (t.arity === "name") {
                n.error("Already defined.");
            }
        }
        this.def[n.value] = n;
        n.reserved = true;
    }
};
```

我们需要一个保留字策略，在一些语言中，用于表示程序结构的词比如 if，就被视为保留字，因此不能用于变量名。词法分析器的灵活性允许我们有更多有用的策略，比如我们可以说在任意函数中，任何符号可用于结构关键字，也可以用于变量，但不能同时成为两者。

当我们想要为函数或代码块创建一个新的作用域时，可以调用下面的函数 new_scope，此函数创建一个 original_scope 的实例。

```
var new_scope = function () {
    var s = scope;
    scope = Object.create(original_scope);
    scope.def = {};
    scope.parent = s;
    return scope;
};
```

优先级

基本的优先级问题是，给定两个运算符之间的操作数，操作数属于哪个运算符？

$$d\,\mathrm{A}\,e\,\mathrm{B}\,f$$

如果大写字符 A 和 B 是运算符，小写字母是操作数的话，操作数 e 是应该绑定在 A 还是 B 上？换句话说，表达式是

$$(d\,\mathrm{A}\,e)\,\mathrm{B}\,f,$$

还是

$$d\,\mathrm{A}\,(e\,\mathrm{B}\,f)\,?$$

为解决此类歧义，让每个 token 都有一个绑定权值 binding powers，对于运算符来说就是优先级，称为绑定权值的原因是它不仅用于运算符，还用于语句中的符号。以及再包含 led 方法和 nud 方法。led（left denotation）方法关注左操作数，如中缀运算符和后缀运算符关注左操作数，所以用 led 方法。nud（null denotation）方法不关注左操作数，比如前缀运算符，数字字面量和字符串字面量的 token 要用 nud 方法。由于字面量这种不关注右操作数的符号也用 nud 方法，所以这就是 nud 不叫 rid（right denotation）的原因——不关注左操作数并不表示就是关注右操作数。一个 token 可以同时拥有这两种方法，比如短横线“—”既可用作前缀运算符表示负号，又可用于中缀

运算符表示减号，因此同时拥有 nud 和 led 方法。注意，led 方法的作用并不是用于获取左操作数，而是用来构建关注左操作数的 token 的语法树，nud 方法也是如此。

表达式

表达式函数 expression 是 TDOP 的核心，它的参数是一个右绑定权值，用来表示对右边“操作数“的绑定能力。强调一下，之前所说的表达式的概念并不限于数学表达式，而是泛指一切语句，因此这里的操作数并不仅指运算符的操作数，而是泛指一切操作对象，操作对象是否是操作数取决于调用 expression 函数的主调方是否是运算符，语句中的关键字（不是运算符）也可以调用 expression。

```
01 var expression = function (rbp) {
02     var left;
03     var t = token;
04     advance();
05     left = t.nud();
06     while (rbp < token.lbp) {
07         t = token;
08         advance();
09         left = t.led(left);
10     }
11     return left;
12 }
```

在 expression 中，第 5 行调用 token 的 nud 方法。nud 方法用于处理字面量、变量和前缀运算符。只要右绑定力小于下一个 token 的左绑定力，就调用下一个 token 的 led 方法。led 方法用于处理关注左操作数的中缀运算符和后缀运算符。nud 和 led 方法中都会递归调用 expression。

expression 会生成一棵语法树，根据调用层次，也可能是语法子树，返回值为这棵树的根。表达式的长度取决于参数 rbp，比如当前的 token 是一个加号，现在调用 expression 来获得加号所绑定的表达式，传入 expression 的参数 rbp 就是加号的右绑定权值，其实就是加号的优先级。因为加号的绑定能力较乘号低，所以被加号绑定的表达式肯定比乘号长，比如 1+2*3+4，第一个加号绑定的表达式为 1+2*3，其中 1 为加号的左子树，2*3 是加号的右子树，如果把加号变成乘号，如 1*2*3，那它能绑定的表达式只能为 1*2。注意，expression 的返回值是 original_symbol 的子类。

给 expression 传递的权值通常是表示运算符的权值，即运算符以自己的权值为参数去调用了 expression，这是通常的且主要的用途。但是词法分析是以 token 为单位，运算符也是 token，只是人为给运算符加了权值，因此任何具有权值的 token（可以不为运算符），如常数等“值”的字面量都可以被 expression 解析，甚至是语法关键字也可以，这也正是 TDOP 可以进行语法分析的原因。expression 是由 nul 和 led 方法组成的，而 led 和 nul 递归调用 expression，相互融合起作用。

注意，如果此时对 expression 没感觉也不要沮丧，下节咱们会再深入分析 expression 的神奇，至少得跟一遍代码才行。

中缀运算符

运算符+是中缀运算符，中缀运算符有左右两个操作数，它关注左操作数，所以它拥有 led 方法。提醒一下，关注左操作数的运算符会有 led 方法，尽管中缀运算符关注左操作数，但 led 方法的作用并不是去获取左操作数，而是构建运算符的语法树，因此运算符中也没有专门去获取右操作数的方法。led 方法把+的 token 变成一个拥有两个分支（first 和 second）的树，first 分支是左操

作数，second 分支是右操作数。左操作数是 led 方法的参数，右操作数需要调用 expression 获取。

```
symbol("+", 50).led = function (left) {
     this.first = left;
     this.second = expression(50);
     this.arity = "binary";
     return this;
};
```

led 方法返回的是 symbol，它也是 original_symbol 的子类。由于 this 包括了左节点 first 和右节点 second，可见 this 就是树根，led 方法返回的是一棵树。

乘号*与+类似，除了 id 和绑定权值不同外，由于*的优先级大于+，所以*的绑定权值更高。

```
symbol("*", 60).led = function (left) {
     this.first = left;
    this.second = expression(60);
     this.arity = "binary";
     return this;
};
```

并不是所有的中缀运算符都如此，但大多数都如此，因此下面可以定义 infix 函数使工作简单一些。infix 函数的参数是符号 id、绑定权值 bp 及一个可选的 led 方法。也就是说没有 led 方法时，infix 提供默认的 led 方法，该 led 方法适用于普通的中缀运算符，如果需要特殊的 led 方法，一定要给 infix 传入专门的 led 方法。infix 就是对 symbol 的封装。

```
var infix = function (id, bp, led) {
     var s = symbol(id, bp);
     s.led = led || function (left) {
          this.first = left;
          this.second = expression(bp);
          this.arity = "binary";
          return this;
     };
     return s;
}
```

下面可以声明一些中缀运算符了。

```
infix("+", 50);
infix("-", 50);
infix("*", 60);
infix("/", 60);
```

===在 JavaScript 中是精确等于运算符（值和类型都要相等）。

```
infix("===", 40);
infix("!==", 40);
infix("<", 40);
infix("<=", 40);
infix(">", 40);
infix(">=", 40);
```

三元运算符拥有 3 个表达式，用？和：分隔，它并不是一个普通的中缀运算符，所以要为其

提供 led 方法。

```
infix("?", 20, function (left) {
     this.first = left;
     this.second = expression(0);
     advance(":");
     this.third = expression(0);
     this.arity = "ternary";
     return this;
});
```

运算符“.”用于选择一个类属性，其右边的 token 必须是标识符，并且会被处理为直接量。

```
infix(".", 80, function (left) {
     this.first = left;
     if (token.arity !== "name") {
          token.error("Expected a property name.");
     }
     token.arity = "literal";
     this.second = token;
     this.arity = "binary";
     advance();
     return this;
});
```

“[”用于动态地从数组中选择元素或对象中选择成员，并且以“]”结束。

```
infix("[", 80, function (left) {
     this.first = left;
     this.second = expression(0);
     this.arity = "binary";
     advance("]");
     return this;
});
```

运算符之间还存在结合性，“左结合”与“右结合”的区别在于同样种类的运算符之间是左边运算符优先级高还是右边运算符优先级高。以上中缀运算符是左结合的，我们同样创建右结合性的运算符，比如短路逻辑运算符，这是通过减少绑定权值实现的。

```
var infixr = function (id, bp, led) {
     var s = symbol(id, bp);
     s.led = led || function (left) {
          this.first = left;
          this.second = expression(bp - 1);
          this.arity = "binary";
          return this;
     };
     return s;
}
```

实现右结合的原理还是很简单的，核心就是 expression(bp - 1)，实参 bp-1 在 expression 函数中就是形参 rbp，其中代码 while (rbp < token.lbp)中的 token 是指下一个运算符，而该运算符的 lbp

就是 rbp+1，即原本与其左边运算符的优先级是相等的，由于已经减 1 了，所以满足循环条件进入循环体，也就是说让右边的运算符比左边运算符先进行了运算，因此实现了右结合性。

下面加入两个逻辑运算符&&和||。如果运算符&&的第一个操作数为假就返回该操作数，否则返回第二个操作数。如果运算符||的第一个操作数为真就返回该操作数，否则返回第二个操作数。

```
infixr("&&", 30);
infixr("||", 30);
```

前缀运算符

前缀运算符是右结合的，我们用于右结合中缀运算符的代码可用于前缀运算符。前缀运算符并没有左操作数，因此没有左绑定权值。有时候前缀运算符也被用于系统保留字。

```
var prefix = function (id, nud) {
     var s = symbol(id);
     s.nud = nud || function () {
          scope.reserve(this);
          this.first = expression(70);
          this.arity = "unary";
          return this;
     };
     return s;
}
prefix("-");
prefix("!");
prefix("typeof");
```

“(” 的 nud 方法需要调用 advance(')')去匹配 “)”，因为 nud 返回内部的表达式，所以 “(” 并不会变成语法树的一部分。

```
prefix("(", function () {
     var e = expression(0);
     advance(")");
     return e;
});
```

赋值操作符

我们可以用 infixr 调用我们的赋值运算符，但我们还希望它能做额外的两件事：确保赋值操作符的左操作数是左值，左值是赋值运算符左边的符号，即可以被赋值的变量；再添加一个 assignment 成员，用以标识这是一个赋值操作符，目的是可以通过此成员快速区分赋值语句。

```
01 var assignment = function (id) {
02      return infixr(id, 10, function (left) {
03           if (left.id !== "." && left.id !== "[" &&
04                          left.arity !== "name") {
05               left.error("Bad lvalue.");
06           }
07           this.first = left;
08           this.second = expression(9);
09           this.assignment = true;
10           this.arity = "binary";
```

```
11              return this;
12          });
13  };
14  assignment("=");
15  assignment("+=");
16  assignment("-=");
```

注意，我们的函数是一系列的封装，assignment 返回的是 infixr 的结果，infixr 返回的是 symbol 的结果。

第 2 行在 infixr 中传入的 led 方法中，第 3～4 行在判断左值是否合法，即是否可写。左值就是赋值运算符=、+=、-=左边的操作数，这种操作数肯定是个变量，形参 left 表示左值，左值只能是 3 种变量:

（1）类属性（域），赋值情况如 OBJ.AGE = 20;

（2）数组元素，赋值情况如 A[1] = 9;

（3）普通变量，赋值情况如 B = A[0]。

对于前两种情况，左值 left 是语法树:

（1）对于类属性，根为 “.”，左子树为 obj，右子树为 *age*;

（2）对于数组元素，根为 “[”，左子树为 *a*，右子树为 1。

这就是第 3 行代码的用意。

常量

函数 constant 在语言中构造常量，常量不关注左操作数，故只有 nud 方法，nud 方法将标识符 token 转换为字面量 token。

```
var constant = function (s, v) {
    var x = symbol(s);
    x.nud = function () {
        scope.reserve(this);
        this.value = symbol_table[this.id].value;
        this.arity = "literal";
        return this;
    };
    x.value = v;
    return x;
};
constant("true", true);
constant("false", false);
constant("null", null);
constant("pi", 3.141592653589793);
```

下面的 symbol("(literal)")是所有数字及字符串字面量的原型，字面量的意义就是本身，所以直接返回。itself 是在之前所说的作用域中定义的。

```
symbol("(literal)").nud = itself;
```

语句

TDOP 中的 expression 适用于所有表达式，函数语言里一切皆表达式，故 TDOP 推导结果适用于所有函数语言。大部分主流语言拥有语句的概念，我们只要在 token 中添加成员 std 即 statement denotation，就可以支持语句了。std 其实就是各语法关键字的语法分析函数，可以见后面的 stmt 函数。

系统保留字肯定有固定的语法，因此必有单独的语法分析函数。函数 statement 一次处理一条

语句，如果当前 token 有 std 方法（此方法是后面用 stmt 方法注册的语法解析函数），说明此 token 应为系统保留字，因此将保存为系统保留字，然后调用其 std 方法进行该关键字的语法分析。为了程序的可读性，statement 中默认只支持赋值和函数调用这两种表达式语法分析。

```
01 var statement = function () {
02       var n = token, v;
03       if (n.std) {
04             advance();
05             scope.reserve(n);
06             return n.std();
07       }
08       v = expression(0);
09       if (!v.assignment && v.id !== "(") {
10             v.error("Bad expression statement.");
11       }
12       advance(";");
13       return v;
14 };
```

statement 是语句分析的入口，所有关键字 token 都有自己的语法分析方法。第 3 行判断如果 token 有自己的分析方法就调用，否则就执行第 8～12 行的语法分析，第 8 行是对赋值语句和函数调用分析，在前面已经通过 assignment 函数把=、+=和-=3 种赋值语句加进去了，也就是说已经为这 3 种符号赋予了绑定权值，故 expression 可以分析它。给 expression 传参为 0 表示最低右绑定能力，即处理完整的表达式（如果不太明白下节再说）。第 12 行调用 advance(';')读入 ";"，这是因为 JavaScript 中的语句是以 ";" 结束，确保了上面所说的：statement 一次解析一个语句。

下面的函数 statements 封装了 statement 函数，是以（end）和 "}" 作为语句分析的界限，也就是可以解析多条语句，因此实现了解析{}中间的代码块。此函数返回一个语句或者语句数组，如果代码块为空就返回 null。

```
01 var statements = function () {
02     var a = [], s;
03     while (true) {
04         if (token.id === ")" || token.id === "(end)") {
05             break;
06         }
07         s = statement();
08         if (s) {
09             a.push(s);
10         }
11     }
12     return a.length === 0 ? null : a.length === 1 ? a[0] : a;
13 };
```

"}" 和(end)是代码块结束标记，第 4 行判断代码块是否结束。第 7 行调用 statement 逐行分析语句，如果语句不为空，就在第 9 行添加到数组 a 中。最后第 12 行根据语句数组的长度返回相应结果。强调，statements 并不负责读取语法块的 "{" 和 "}"，它只负责处理中间的语句。连同{多个语句}一同处理的是后面的函数 block，它专门处理代码块。

下面的函数 stmt 用于添加语句符号到符号表中，参数 s 表示语句 id，参数 f 表示 std 函数，也就是把 std 注册为 id 的语法分析函数。

```
var stmt = function (s, f) {
     var x = symbol(s);
     x.std = f;
     return x;
};
```

语句块是用大括号括起来的一组语句，大括号相当于语句块的作用域。(JavaScript 并没有块作用域，简单 JavaScript 修正了这个问题。)

```
1 stmt("{", function () {
2       new_scope();
3       var a = statements();
4       advance("}");
5       scope.pop();
6       return a;
7 });
```

用 stmt 注册 token'{'的语法分析函数，此 function 函数是 token'{'的语法分析函数 std。可见第 4 行代码 advance("}")，说明代码块的结束边界“}”是由“{”的语法分析函数读取的，说明了函数 statements 只负责其中的语句块。

下面的函数 block 用于处理代码块。

```
1 var block = function () {
2       var t = token;
3       advance("{");
4       return t.std();
5 };
```

其中第 3 行调用的 advance('{'}函数确保当前 token 是“{”，t 即是当前 token，第 4 行调用 t 的 std 方法，即上面为“{”注册的 std 方法。同时第 3 行代码说明代码块的起始边界“{”是由 block 读取的。提醒，“}”是由“{”的语法分析函数读取的，可见上面为“{”注册的 std 函数实现。

下面的 var 语句在当前块作用域中定义一个或多个变量，各个标识符后面有一个可选的=作为初始化表达式。

```
01 stmt("var", function () {
02       var a = [], n, t;
03       while (true) {
04          n = token;
05          if (n.arity !== "name") {
06                n.error("Expected a new variable name.");
07          }
08          scope.define(n);
09          advance();
10          if  (token.id === "=") {
11                t = token;
12                advance("=");
13                t.first = n;
14                t.second = expression(0);
15                t.arity = "binary";
```

```
16                a.push(t);
17          }
18          if (token.id !== ",") {
19                break;
20          }
21          advance(",");
22      }
23      advance(";");
24      return a.length === 0 ? null : a.length === 1 ? a[0] : a;
25  });
```

第 1 行注册关键字 var(变量定义)的语法分析程序，即 token var 的 std 程序是此处的 function。第 5 行确保 token 必须为标识符才可以为变量名。第 8 行把标识符加入当前作用域，用于统计符号信息以判断是否重复定义。第 9 行调用 advance 移进下一个 token 后，在第 10 行判断当前 token 是否是可选的=，如果是=，说明是在初始化，第 14 行调用 expression 分析=后面的赋值表达式。多个变量之间是用逗号分隔的，若没有逗号则说明变量的声明结束，第 18 行判断是否还有变量。

下面的 while 语句定义一个循环，包括放在括号中的条件表达式和代码块。

```
1 stmt("while", function () {
2     advance("(");
3     this.first = expression(0);
4     advance(")");
5     this.second = block();
6     this.arity = "statement";
7     return this;
8 });
```

第 1 行注册关键字 while 的语法分析程序，即 token while 的 std 程序是此处的 function。第 2 行读入条件表达式左边的小括号 “(”。第 3 行分析 while 后面的条件表达式，传给 exression 的权值参数为 0 表示分析整个表达式。第 4 行读入条件表达式右边的小括号 “)”。第 5 行调用 block 解析 while 的循环体。

下面的关键字 “if” 用于条件执行，在代码块后若看到了关键字 else，就继续判断是否是 else if。

```
01 stmt('if', function() {
02      advance('(');
03      this.first = expression(0);
04      advance(')');
05      this.second = block();
06      if (token.id === 'else') {
07            scope.reverse(token);
08            advance('else');
09            this.third = token.id === 'if' ? statement() : block();
10      } else {
11            this.third = null;
12      }
13      this.arity = 'statement';
14      return this;
15 });
```

选择分支中的 else 部分支持 else 和 else if 两类。第 6 行发现 else 后，顺便将 else 添加到作用域中表示 else 为保留字，可见保留字也是普通符号，只不过是提前把它定义为符号，由于不允许重复定义符号，导致用户就无法再次定义该符号，从而实现了使符号成为保留字。第 8 行通过 advance("else")跳过 else，此时当前 token 应该是 if 或 “{” 或单条语句或语句块的起始 “{”，如果是 if，说明正是 “else if”，因此会调用 statment 去做语法分析，否则只是 else 后跟语句块即'{'block'}'，则调用 block 做语法分析。

下面的 break 语句用于跳出循环。

```
stmt("break", function () {
     advance(";");
     if (token.id !== ")") {
         token.error("Unreachable statement.");
     }
     this.arity = "statement";
     return this;
});
```

下面的 return 语句用于从函数返回，带有一个可选的表达式作为返回值。

```
stmt("return", function () {
     if (token.id !== ";") {
          this.first = expression(0);
     }
     advance(";");
     if (token.id !== ")") {
         token.error("Unreachable statement.");
     }
     this.arity = "statement";
     return this;
});
```

函数

函数是可执行的对象，为了可递归调用，函数有一个名字，用小括号括起来的参数列表和用大括号括起来的代码块。参数有自己的作用域。

```
prefix("function", function () {
     var a = [];
     new_scope();
     if (token.arity === "name") {
         scope.define(token);
         this.name = token.value;
         advance();
     }
     advance("(");
     if (token.id !== ")") {
         while (true) {
             if (token.arity !== "name") {
                 token.error("Expected a parameter name.");
             }
             scope.define(token);
```

```
                a.push(token);
                advance();
                if (token.id !== ",") {
                    break;
                }
                advance(",");
            }
        }
        this.first = a;
        advance(")");
        advance("{");
        this.second = statements();
        advance("}");
        this.arity = "function";
        scope.pop();
        return this;
});
```

函数是用“(”调用的，它可以携带 0 或多个逗号分隔的参数，我们要判断它的左运算对象即函数名、function 等是否可作为可调用的函数。注意，函数调用是用 infix 实现的，因为“(”关注左操作数即函数名，而函数定义是用 prefix 实现的，因为是绑定在关键字 function 上，function 不关注左边操作数，只关注右边的操作数即函数体。

```
01 infix("(", 80, function (left) {
02       var a = [];
03       if (left.id === "." || left.id === "[") {
04            this.arity = "ternary";
05            this.first = left.first;
06            this.second = left.second;
07            this.third = a;
08       } else {
09            this.arity = "binary";
10            this.first = left;
11            this.second = a;
12            if ((left.arity !== "unary" || left.id !== "function") &&
13                      left.arity !== "name" && left.id !== "(" &&
14                      left.id !== "&&" && left.id !== "||" && left.id !== "?") {
15                 left.error("Expected a variable name.");
16            }
17       }
18       if (token.id !== ")") {
19            while (true)  {
20                a.push(expression(0));
21                if (token.id !== ",") {
22                      break;
23                }
24                advance(",");
25            }
26       }
27       advance(")");
```




```
28      return this;
29 });
```

因为函数调用的语法解析是用 infix 中置运算符来注册的，因此这个 function 函数是符号“(”的 led 方法。这同时说明“(”认为 left 是其左边的操作数，因此 left 肯定是函数名的 token。后面会生成以“(”为根的二叉树，左节点是函数名，右节点是实参。

第 2 行数组 a 用来存储实参。第 3～7 行处理三个节点的情况，如对象调用方法。说下第 8～17 行处理两个节点的情况，这是一般的情况更具代表性。第 9 行使 arity 等于 binary，标明了根节点“(”的类型为二元运算符，因此知道“(”是根节点，它下面还有两个节点。第 10 行使左操作数 left 作为“(”的左节点。第 11 行用来存储实参的数组 *a* 作为“(”的右节点。第 12 行判断函数调用的合法性，不深究了。第 18～26 行用于读取以逗号分隔的实参。读取完成后，第 27 行读入实参右边的“)”。最后返回自己，即以“(”为根的子树。

关键字 this 是特殊的变量，在方法调用中它指向对象。

```
symbol("this").nud = function () {
    scope.reserve(this);
    this.arity = "this";
    return this;
};
```

数组直接量

数组直接量是一组用中括号括起的 0 个或多个逗号分隔的表达式，各个表达式的结果被收集到新数组。数组定义形式：var a = [元素 1，元素 2，...，元素 *n*]

```
prefix("[", function () {
    var a = [];
    if (token.id !== ")") {
        while (true) {
            a.push(expression(0));
            if (token.id !== ",") {
                break;
            }
            advance(",");
        }
    }
    advance(")");
    this.first = a;
    this.arity = "unary";
    return this;
});
```

对象直接量

对象直接量是由一对大括号括起的用逗号分隔的键值对儿组成，键值对儿中的键和值是用冒号分隔，键要么是字面量，要么是可以被处理为字面量的标识符。

```
01 prefix("{", function () {
02     var a = [];
03     if (token.id !== ")") {
04         while (true) {
05             var n = token;
06             if (n.arity !== "name" && n.arity !== "literal") {
```

```
07                    token.error("Bad key.");
08                }
09                advance();
10                advance(":");
11                var v = expression(0);
12                v.key = n.value;
13                a.push(v);
14                if (token.id !== ",") {
15                    break;
16                }
17                advance(",");
18            }
19        }
20        advance(")");
21        this.first = a;
22        this.arity = "unary";
23        return this;
24    });
```

对象直接量形如 var *a* = {key1:value1,key2:value2,...,keyn:valuen}。

第 2 行数组 *a* 用于存储对象直接量中值的语法树（依然是 original_symbol 子类），key 记录在对应 value 的语法树中的 key 属性中，即数组 *a* 的元素是含有 key 的 value 的语法树。第 3～19 行循环处理键值对儿，此时 token 为 key，在第 5 行用 *n* 来备份这个 key，后面要将此 key 作为属性添加到 value 的树结构中。第 9 行跳过 key，使当前 token 为 ":"。第 10 行确认当前 token 是 ":"，然后读入下一个 token，当前 token 应该是 value。第 11 行解析 value 的表达式。expression 的返回值是 original_symbol 的子类，因此第 11 行可以为其添加属性 key。第 12 行使 key 等于 n.value，*n* 是之前 key 的 token，上面第 5 行用 n 备份了 key 的 token。而 value 是个表达式，expression 返回的是 value 的语法树（此时并不知道数值是多少，因为计算过程与此时的语法分析无关，需要后期执行时遍历树结构才可算出真正的 value）。总之变量 *v* 是 value 的语法树，在该树中加了个 key 属性记录此 value 的 key。第 13 行把包含 key 的 value 添加到数组 *a* 中。到了第 16 行完成了一对键值的读取，第 17 行读取键值对儿后的 ","。最后读取 "}"，返回对象直接量。

待做的工作及展望

语法树可以被代码生成器解析，或者由解释器解析。只需要非常少的计算就能够生成语法树，正如我们看到的，写一个创建语法树的程序不费什么力气。

我们可以让 infix 函数接受操作码，它有助于代码生成。

我们还可以支持更多的语句（比如 for、switch 和 try）、标签、更多的错误检查、错误恢复以及更多的运算符。

我们也可以让语言更具扩展，定义新变量同样容易，我们可以让程序员添加新的运算符和语句。

原文结束。如果没搞懂 TDOP，马上进入下节。

5.3 TDOP 原理

5.3.1 一些概念

常见的语法分析方法有自上而下即自顶向下分析法，英文缩写为 TD（Top Down），是通过自上而下递归下降推导的方式进行语法分析。还有算符优先分析法，英文缩写为 OP（Operator



Precedence），是通过比较算符优先级大小进行自下而上归约的方式进行语法分析。将这两者结合起来就是 TDOP——自上而下算符优先分析法。提醒，算符不仅包括运算符，而且是广义的，泛指文法中的终结符。算符就是具有操作数的操作符，运算符是操作符的特例，操作数是数学表达式的操作符就是运算符，本书中多数用操作符涵盖所有算符情况。

不同的操作符有不同的优先级，比如乘法运算符*的优先级就比加法运算符+的优先级高，我们遇到 1+2×3 时肯定是先做乘法 2×3，使 2×3 的结果作为+的另外一个操作数与 1 相加。人能轻易判断出优先级，但人如何让计算机也能判断呢？首先得让计算机理解什么是优先级。

优先级的产生源自于操作数不知道属于哪个操作符。以下面的表达式为例，其中大写字母是操作（运算）符，小写字母表示操作数。

$$a\ \mathrm{B}\ c\ \mathrm{D}\ e$$

问题来了，c 是和 a 一起执行 B 操作即$(a\ \mathrm{B}\ c)\ \mathrm{D}\ e$，还是 c 和 e 一起执行 D 操作即 $a\ \mathrm{B}\ (c\ \mathrm{D}\ e)$ 呢？中间的 c 表示很为难。运算是由“操作符和操作数”组成的，操作符在位置上紧挨着操作数，因此对于优先级的直观理解就是，对于同一个操作数，它周围的哪个操作符更加“吸引”它，它就先跟哪个操作符“优先”计算，这就是操作符优先级的由来。

更吸引操作数的操作符在运算上就更优先——这看似是我们本能上的理解，其实这是数学老师给“写死”在我们大脑中的硬编码，今后在遇到表达式计算时我们才能正确地先去计算优先级更高的运算符。数学中看似无限灵活的计算推导都源自于那几条规定好的硬编码。在计算机中如何体现优先级呢？只能采取同数学老师一样的办法，我们要给计算机先规定好一些操作符的优先级，即刚才所提到的对操作数的“吸引能力”，这种吸引力的术语被称为绑定权值 bp（bind power）。操作符的 *bp* 表示操作符的优先级，数值越大越优先。

当一个操作数的两边都有操作符时才是操作符绑定权限发挥作用的地方，即操作数 c 是跟操作符 B 操作还是跟操作符 D 操作，取决于操作符 B 和 D 的 *bp*，谁大就跟谁。

根据操作符在操作数“们”中的位置来划分可分为前置操作符（比如!）、中置操作符（比如×）及后置操作符（比如++），前面强调了操作符“们”，显然只有中置操作符在操作数“们”中间，因此中置操作符有多个操作数，比如×就有两个操作数，?:就有 3 个操作数。中置操作符同时吸引左右两边的操作数，对两边操作数的吸引力是一致的，如果 B 和 D 都是中置运算符即都有两个操作数，比如表达式 $a\ \mathrm{B}\ c\ \mathrm{D}\ e$，从字面上看 c 既是 D 的左操作数，又是 B 的右操作数，c 在抉择操作符时，是在判断 B 更关心右边的操作数（c 自己）还是 D 是否更关心左边的操作数（c 自己），这可以用操作符 B 和 D 的 *bp* 值来决定。可是前置和后置操作符都只有一个操作数，其优先级比中置运算符高，当它们和中置运算符一同混在表达式中却容易出问题，比如表达式-1*2 中的-此时不再是中置运算符，而是前置运算符，其优先级就比*高。也就是说，此情况中*关心左边的操作数，而-号由于是前置运算符，它并不关心左边的操作数，只关注它右边的操作数。换句话说，如何让计算机知道此情况中的-是前置运算符而不是中置运算符呢？一种解决思路是，我们只要让计算机知道操作符是否吸引左边的操作数就可以区分出各种类型的操作符。算法中我们按照“操作符”是否关心吸引左边的操作数为每个运算符添加一个 nud 方法和 led 方法，含义如下：

nud（null denotation）不关心（吸引）左边的操作数

led（left denotation）关心（吸引）左边的操作数

强调，只有操作符才有吸引操作数的能力，因此 nud 和 led 方法存在于操作符（包括运算符）中。

为实现一个通用算法，不仅是让运算符包括 nud 和 led 方法，而是让所有符号都包括它们，上节示例中的符号 original_symbol 就是这个作用。



```
var original_symbol = {
    nud: function() {
    ...
    }
    led: function(left) {
    ...
    }
}
```

用 nud 方法的符号：因为常量等非操作符是不关心操作数的，也就不关心左边的操作数，因此变量、字面量等和前置运算符用 nud 方法。

用 led 方法的 token：中置运算符和后置运算符都关心左边的操作数，因此用 led 方法。

一个符号可以同时具备 nud 和 led 方法，比如-可以既是前置运算符表示负号，又可以是中置运算符表示减号。

表达式函数 expression 是 TDOP 技术的核心。它的参数是一个右绑定权值，强调一下是“右”绑定权值。表达式语法树的构建过程是 expression 与 led、nud 之间的递归调用完成的，对大多数人来说容易迷惑，这也正是 TDOP 的难点。TDOP 是 expression 和 led 与 nud 之间的递归调用，递归的起始入口是 expression 函数。

5.3.2　一个小例子

表面看上去 expression 是专门用来解析数学表达式，也就是运算符和操作数组成的表达式，但其功能远不止如此，原则上只要符号包括 nud 或 led 方法就可以被 expression 处理，因此一般情况下符号不仅包括运算符，还包括各种用于“定义”的符号。换句话说，要想使用 expression 解析表达式结果，就要为表达式中的所有符号准备 nud 或 led 方法还有绑定权值。为了便于理解，先将表达式的概念退化成简单的数学表达式。另外，后面也会用数学表达式来理解 TDOP，因此会不断地提及运算符，但大家要清楚它只是操作符的一种，在 TDOP 中操作符泛指具有绑定权值的符号。

expression 的形参是 rbp，即右绑定权值，也就是说 expression 的调用者应该是个运算符，只有运算符才有操作数的需求。运算符用自己的绑定权值作为 rbp 调用 expression，目的是用来解析该运算符的右操作数。因为中缀运算符对左右两个操作数的吸引力是一样的，所以可用 lbp 表示 rbp。符号创建函数 symbol 的参数 bp 是给符号的 lbp 赋值的，前缀符号中并没有 rbp，如何表达前缀运算符对右操作数的吸引能力呢？解决的方案是在 nud 方法中用较大的数字作为 rbp 参数调用 expression，具体多大取决于前缀运算符的优先级，比如 expression(80)，具体可回顾之前的 prefix 实现。

对于表达式 1+2+3：左边第一个+的左操作数是 1，假设现在是为此加号获取右操作数，+的绑定权值是 50，那么调用 expression 的参数就为 50，即 expression(50)。expression(50)会获取到 2，2 将成为+的右操作数。详细过程下面说，现在希望大家有这样的共识：expression 的调用者是运算符，哪个运算符想获得右操作数，哪个运算符就以自己的绑定权值作为 expression 的参数右绑定权值 rbp 调用 expression。切记，expression 是为调用者（运算符）获得其右操作数，根据运算符的优先级，其右操作数可能是具体的一个数，也可以是一个子表达式的结果。比如 1+2×3，那么加号+的右操作数就是 2×3 的结果。

把两个关键方法贴出来，如表 5-1 所列，这里是用全局的行号，各函数的行号以左侧列为准。开始分析 1+2*3，已知当前 token 是 1。



表 5-1

```
01  var expression = function (rbp) {
02      var left;
03      var t = token;
04      advance();
05      left = t.nud();
06      while (rbp < token.lbp) {
07          t = token;
08          advance();
09          left = t.led(left);
10      }
11      return left;
12  }
```

```
symbol("+", 50).led = function (left) {
    this.first = left;
    this.second = expression(50);
    this.arity = "binary";
    return this;
};
```

```
symbol("*", 60).led = function (left) {
    this.first = left;
    this.second = expression(60);
    this.arity = "binary";
    return this;
};
```

初始调用是 expression(0)，即 rbp 是 0，因为 expression 是由运算符调用的，它的 rbp 参数是该运算符的右绑定权值，也就是吸引右操作数的能力。是谁想从表达式 1+2×3 获取自己的右操作数呢？看似是空的，没有运算符，因此 rbp 就为 0？不对，这并不是 expression 的 rbp 为 0 的原因，rbp 为 0 是为了在 expression 中的 while 循环处获得最大的表达式处理长度，0 肯定小于任何符号的绑定权值，这样一来，整个表达式 1+2×3 都将被完整解析。

此时进入第一层 expression，token 为 1。

expression 函数第 3 行用 t 备份了 1 的 token，第 4 行使 token 变为+，第 5 行调用 1 的 nud 方法，它是常量，会返回 1 本身，即 left 就是 1。此时 rbp 为 0，token.lbp 为 50，因此进入循环体。第 7 行使 *t* 成为+，第 8 行使 token 变成 2，第 9 行调用+的 led 方法，参数 left 是 1。

自此进入'+'.led 的第 2 行，把 1 作为+的左子树，第 3 行继续调用 expression，参数为+的 lbp，即 50。

此时进入第二层 expression，rbp 为 50，token 为 2。

第 3 行使 *t* 为 2，第 4 行使 token 为×，第 5 行使 left 为 2。第 6 行 rbp 为 50，token.lbp 为 60，条件满足进入循环，第 7 行使 t 成为×，第 8 行使 token 成为 3，第 9 行调用'*'.led，left 为 2。

自此进入'×'.led 的第 8 行，把 2 赋值给×的左子树，调用 expression(60)，60 是'×'的 lbp。

此时进入第三层 expression，rbp 为 60，token 为 3。

第 3 行使 *t* 成为 3，第 4 行使 token 变成结束 end，end 之前是用 symbol("(end)")添加的，它的 lbp 采用的是默认值 0。第 5 行使 left 为 3，第 6 行 rbp 是 60，token.lbp 是 0，条件不成立，因此第 11 行返回 3。

此时退出第 3 层 expression，回到×.led 的第 10 行。

把 3 赋值给右子树，第 11 行返回以×为根，2 为左子树，3 为右子树的语法树。

此时退出'×'.led，回到第二层 expression 的第 9 行。

使 left 成为×的语法树。继续第 6 行判断条件，rbp 为 50，token 为 end，token.lbp 为 0，条件不成立，第 11 行返回×的语法树。

此时退出第二层 expression，回到'+'.led 第 3 行。

使右子树为×的语法树，第 5 行返回以+为根的语法树，左子树为 1，右子树为×的语法子树。

此时退出'+'.led，回到第一层 expression 的第 9 行。

使 left 成为+的语法树，回到条件第 6 行，rbp 为 0，token 是 end，其 lbp 为 0，条件不成立，因此执行第 11 行返回+的语法树，到此结束。

5.3.3 expression 的思想

首先说明，expression 并不用于语法分析，TDOP 中负责语法分析的部分指的是其中的“TD”——

自上而下语法分析，我们要额外写一些语法分析函数去进行语法分析。expression 属于 TDOP 中的“OP”——算符优先，它只用于解析表达式结果，当语法分析函数中需要计算表达式时就调用 expression。此处的表达式不仅包括数学表达式，还包括一些语法关键字组成的有关定义的表达式。expression 还是有点难度的，为容易理解，下面用数学表达式来解释 expression 的原理。

我们在计算表达式时，都是按照优先级去计算的，先计算优先级更高的运算符的表达式，然后再用其结果作为较低优先级运算符的操作数，比如表达式 1+2×3，先计算的肯定是 2×3，用其结果 6 作为+的右操作数。其实人是从左到右扫描完整个表达式时，拿+、−、×、/来说，发现了最高级的×和/，决定先做×和/的运算，再将这两个运算结果分别作为+和−的操作数。其实根本用不着事先扫描完整个表达式，因为我们在选择最高优先级的运算符计算时，我们是在判断操作数隶属于哪个运算符，也就是说操作数是夹在两个运算符之间时才需要决判。如果操作数不在运算符之间，其两侧只有一个运算符，肯定就不需要择优了，比如操作数在表达式的首尾位置时，如 1+2×3−4/5 中的 1 和 5，根本不需要判断它们属于哪个运算符，因为各自两侧只有 1 个运算符。

因此，操作数两侧有两个运算符时我们才需要择优（即找出优先级更高的运算符，将该操作数作为更高优先级的运算符的操作数）。操作数只隶属于紧邻其两侧（其左和其右）的运算符之一，要么成为左边运算符的右操作数，要么成为右边运算符的左操作数，肯定是隶属于这两者优先级更高的那个，与整个表达式中优先级最高的那个运算符是没有关系的，完全没必要考察完整个表达式。因此只要一一考察操作数两侧的运算符的优先级就行了，这正是 expression 的本质：每次让两个运算符争抢它们中间的操作数。

expression 函数原理就是从左到右分析表达式，以运算符为中心的左右操作数的两两处理，判断夹在两个运算符之间的操作数到底属于哪个运算符的对象。比如表达式 *a+b−c−d+e*，其处理过程是先以 0 值的 rbp 调用 expression(0)以启动解析过程。用 0 的原因是为了处理完整个表达式，后面会说原因。expression 处理表达式的逻辑是把运算符两两比对优先级，下面以竖线|XxX|为每次处理的分组，其中大写字母是运算符，小写字母是操作数。表达式 *a+b×c−d/e* 在 expression 中的处理过程是：

```
|假想运算符 a+|b-c-d+e
a|+b-|c-d+e
a+b|-c-|d+e
a+b-c|-d+|e
```

4 个分组中，每次都是两个运算符在抢中间的操作数。

还是通过例子去理解，以最简单的中置运算符为例来说明 expression 处理表达式 aSwTeUg 的流程，其中小写字母是操作数，大写字母是运算符。注意，这里说的是中缀运算符，以下情况完全针对中缀运算符，不包括前缀，后面你会看到差别。

（1）假设已处理了一部分，当前 token 为操作数 *w*。

进入 expression 时当前 token 为操作数 *w*，由于涉及优先级，操作数 *w* 既可能是下一个运算符 T 的左操作数，也可能是上一个运算符 S 的右操作数。为了确定操作数 *w* 属于谁，需要比较这两个运算符的优先级，谁的优先级大就表示谁吸引操作数的能力强，那么操作数 *w* 就归属谁。在进入 expression 之前，必须以运算符 S 自己的绑定权值作为参数去调用 expression(S.lbp)。提醒!中置运算符对左右两个操作数的绑定能力是一样的，所以可用 lbp 当成 rbp 传给 expression。进入 expression 后通过 advance 获得运算符 T，然后拿自己即运算符 S 去和运算符 T 比较优先级，这就是下一步的工作。



（2）获得运算符 T。

对于中缀运算符来说，刚进入 expression 时当前 token 肯定为操作数 *w*。后面调用了两次 advance，第一次调用 advance 时是在循环体外，使当前 token 为运算符 T。

（3）比较运算符 S 和运算符 T 的优先级。

在 expression 中 while 循环的条件处，拿运算符 S 的优先级(即 expression 的形参 rbp)去和运算符 T 的优先级比较，分以下两种情况。

① 若运算符 S 的优先级大于等于运算符 T 的优先级，就直接返回 left 即操作数 *w*，操作数 *w* 是运算符 S“右边的”操作数，也就是说这时它才被承认是运算符 S 的右操作数。到此完事儿，下面不用执行了。

② 若运算符 S 的优先级小于运算符 T 的优先级，这说明运算符 T 的吸引操作数的能力更强，既然运算符 T 是中置运算符，说明它左右两边各有一个操作数，那么 left 即操作数 *w* 就要被“吸走”成为运算符 T 的左操作数（没有机会成为运算符 S 的右操作数了）。不过这才完成左操作数，运算符 T 还需要右操作数，于是进入下一步获取其右操作数。

（4）获得运算符 T 右边的操作数，即操作数 *e*。

接着在 expression 中 while 循环体内第二次调用了 advance，使当前 token 变成了运算符 T 右边的操作数 *e*。同样遇到了前一个运算符 S 碰到的优先级问题，运算符 T 右边的操作数 *e* 未必是运算符 T 的右操作数（因此这里说成“右边的操作数”较为严谨）。为此依然要套用步骤 1 的方法去和下一个运算符 U 比较优先级。但是这里并不是直接递归调用 expression，而是通过运算符 T 的 led 方法间接调用 expression，因为我们要通过 led 方法实现语法树。

（5）调用运算符 T 的 led 方法去建立以运算符 T 为根的语法树。

因为在 expression 中调用 T.led 前已经先调用了 advance，使 token 更新为操作数 *e*，接着以 left 即操作数 *w* 为参数调用 T.led。T.led 用于建立一棵树，参数 left 作为运算符 T 的左操作数，即左子树。前面说过，和运算符 S 遇到的问题一样，由于优先级的问题，运算符 T 右边的操作数 *e* 未必会先跟运算符 T 进行计算，万一操作数 *e* 后面还跟着优先级更高的运算符 U，操作数 *e* 就得先作为运算符 U 的左操作数先参与运算符 U 的计算。因此操作数 *e* 不能草率地认为就是运算符 T 的右节点即右操作数。

到此，以下的操作开始重复步骤（1）～（5）。

两种情况下递归会结束。

（1）expression 中 while 循环条件不满足时，即操作数左边运算符的优先级不比右边运算符的优先级低。

（2）表达式被解析完成，没有多余的符号时，表示结束符号的绑定权值为 0，本质上还是属于第一种情况。

以上 aSwTeUg 的流程在代码中说明更清楚一些，强调，这是对于中缀运算符的例子。注释都加了下划线，源码用粗体显示。

```
var expression = function (rbp) {
//expression 是由运算符 S 调用的，对于中缀运算符来说，此时当前 token 为操作数 w。
    var left;   //此 left(操作数 w)是左操作数的意思，即只有作为运算符 T 的操作数时才是真正的左操作数，万一下面即将获得的运算符 T 的优先级(绑定权值)不比上一个运算符 S 的优先级高，此 left 就是运算符 S 的右操作数，因此还是叫操作数 w 更严谨。
    var t = token;  // 获得操作数 w
    advance();      // 获得运算符 T，它就是下面循环中 t.led 中的 t
    left = t.nud();    // 获得操作数 w 的值
    while (rbp < token.lbp) {  //rbp 是上一个运算符 S 对其右操作数的绑定力，token.lbp 是运算符
```

```
T 对其左操作数的绑定能力。若运算符 S 的优先级大于等于现在的运算符 T 的优先级，说明操作数 w 应该属于运算符
S，即属于运算符 S 的右操作数，因此不进入循环，直接返回 left（即操作数 w）给运算符 S 作为它的右操作数。
若运算符 S 的优先级小于运算符 T 的优先级，说明确实运算符 T 吸引操作数的能力更强，故操作数 w 应该作为运算
符 T 的左操作数，此时的 left（即操作数 w）才是名副其实的“左”操作数
        t = token;  // 用 t 备份当前 token 即运算符 T
        advance();  // 使当前 token 为运算符 T 右边的操作数 e
        left = t.led(left);  // 使左操作数 left（即操作数 w）作为参数调用运算符 T 的 led 方法，去
构建以运算符 T 为根，以 left（即操作数 w）为左节点的语法树（其右节点还是要递归调用 expression 去解析，
可见 led 的实现）
    }
// 执行到此，无论运算符 S 和运算符 T 谁的优先级高，此时的 left 已经是运算符 S 的右操作数了，left 要么就
是操作数 w，要么就是以运算符 T 为根的语法树
    return left;
}
```

再次强调，expression 只用于解析表达式结果，并不用于语法分析，它只负责调用符号的 nud 或 led 方法，至于 nud 或 led 方法中是否涉及语法分析的功能，那是该符号自己协调的事，在 expression 看来，它只是调用符号的 led 或 nud 方法去获得该符号的结果而已，也许获得结果的过程中做了语法检查，这也很正常。

5.3.4　while（rbp < token.lbp）的意义

为方便观察再贴一次 expression 代码。

```
01 var expression = function(rbp) {
02     var left;
03     var t = token;
04     advance();
05     left = t.nud();
06     while (rbp < token.lbp) {
07         t = token;
08         advance();
09         left = t.led(left);
10     }
11     return left;
12 };
```

expression 是 TDOP 的核心，第 6 行的 while 循环是 expression 的核心。表达式分析是从左到右进行的，此循环的直接目的是判断左边已分析过的子表达式即 left 是否为下一个运算符的左操作数，也就是前面部分表达式的语法树是否作为下一个运算符的左操作数，因为 left 始终作为下一个运算符 t 的参数传进 t.led，并且总被更新为从 t.led 中返回的表达式的结果（语法树）。

正是由于第 6 行 while 循环的条件，expression(rbp)中的 rbp 的值决定了表达式可被分析的长度（可以走多远），由于 expression 是递归调用的，以下假设是在同一调用层：rbp 始终是调用 expression 的运算符的绑定权值，token 却是表达式中多个运算符，在调用 t.led 前，token 为运算符 T，由于 t.led 中会继续调用 expression，在那层的 expression 同样会执行 advance 使全局变量 token 更新，当从 t.led 返回后，token 的值已经不再是当初的运算符 T 了。while (rbp < token.lbp)中的 token 到底指向了多少个运算符，取决于 token.lbp 是否大于 rbp。比如对于表达式 1+2×3−4 来说，+、−和×的绑定权值分别是 9、9 和 10，expression(9)调用的话，rbp 为 9，token.lbp 为 9，while 循环不执行，因此只会返回 1。若以 expression(0)调用，rbp 为 0，无论递归调用了 expression 多少层，



在首层 expression 中 rbp 的值是 0，无论 token 指向的是什么运算符，0 永远小于所有 token.lbp ，那么会解析完整个表达式。

TDOP 算符优先有些像深度优先，优先级更高的运算符会优先计算，因此在树中更具深度。

5.3.5 进入 expression 时当前 token 的类别

进入 expression 时，当前 token 未必是操作数，很明显，这取决于表达式的运算符类型。之前为理解简单，都是以中缀运算符举例，如 1+2×3 进入 expression 时当前 token 是操作数。对于前缀操作符，如!2+3×4 进入 expression 时当前 token 肯定是前缀运算符。也就是说，进入 expression 时，当前 token 肯定是表达式的首符号。

无论是 nud 还是 led 方法，根据 expression 的结构设计，调用 nud 和 led 方法之前，肯定要先调用 advance 更新 token，进入 nud 或 led 方法时，当前 token 一定是表达式中调用 nud 或 led 的符号的右边的符号，即这两个方法所属符号右面的符号，所以前一个 token 是这两个方法对应的符号。如 1+2，进入'+'.led 时，当前 token 是 2，前一个 token 是'+'，理解这个很重要，我们在程序中经常用到。

以下面这两个表达式举例。

```
表达式 1:   !0+1
表达式 2:   1+!0+2
```

我们已经熟悉 expression 函数的结构了，expression 的功能是为调用者获得其右操作数，以 while 循环为界，功能分为两步实现。

（1）在 while 循环之前，获得 expression 调用者的右边的操作数，这是用 nud 方法来实现的，因为 nud 就只关注右操作数。

（2）while 循环负责判断第 1 步中获得的调用者的右边的操作数是否为该调用者的右操作数，万一是比后面优先级更高的运算符的左操作数呢，这是用 led 方法实现的，它关注左操作数。返回的结果是比该优先级更高的运算符的结果，使该结果作为 expression 调用者的右操作数（这次是真正的右操作数，不再是右边的操作数了）。

注意！expression 中有两处调用了 advance()，分别是在循环体外的 nud()和循环体内的 led()之前，为什么一定要在这两个函数之前调用 advance 以获得下一个 token 呢？顺序调换行吗？肯定是不行的，advance 使 token 指向下一个运算符或操作数，相当于为 nud 和 led 提供了“参数”。刚进入 expression 时，对于表达式 1 来说，当前 token 是运算符!，对于表达式 2 来说，当前 token 是操作数 1，尽管两个表达式的当前 token 类型在进入 expression 之初是不同的，但在 expression 中的循环 while (rbp < token.lbp)处，当前 token 的类型一定得是运算符才行，token 一定是需要左操作数的运算符或符号，因为在循环体中要调用 led 方法，它需要左操作数。为保证这一点，必须使 while 循环上面的 nud()在执行过后，当前 token 至少是需要左操作数的运算符，因此中缀运算符也行。对于调用 nud 方法的符号有两种。

（1）要么是不需要右操作数，只需要返回其值的字面量，也就是不需要在 nud 中继续递归调用 expression 以获得其不需要的右操作数。

（2）要么是需要右操作数的前缀运算符，需要递归调用 expression 以获得其右操作数。

对第一种情况，在 nud 之前调用的 advance 是使当前 token 变成了字面量右边的运算符，即 while (rbp < token.lbp)中的 token。对第二种情况，在 nud 之前调用的 advance 是使当前 token 变成了前缀运算符右边的操作数（这就是上面所说的，advance 为 nud 提供了参数，因为 advance 后，token 变成了前缀运算符右边的操作数，前缀运算符.nud 需要该操作数），该 nud 方法中必须是要



递归调用 expression 去获得右操作数才行，并且从 expression 返回后，肯定是使当前 token 变成运算符，token 绝不会是操作数，因为从 expression 的返回条件肯定是在 while (rbp < token.lbp)中因为循环条件不成立而返回，也就是绑定权值的大小不满足条件，而只有运算符才有绑定权值，所以在 nud 中调用的 expression 保证了当前 token 必是具有绑定权值的运算符。(在前缀运算符的 nud 方法中调用 expression 获得前缀运算符的右操作数，进入该层的 expression 时，当前 token 就是操作数，其后的递归就是按照之前介绍的中缀运算符流程来处理。)

总之，以 while 循环为界，在语句 while (rbp < token.lbp)处，token 必须为运算符（只要有绑定权值就认为是运算符，是广义的，包含语法关键字）。由于 while 循环前只有一个函数，即 nud，因此为保证执行 while 循环前 token 一定是运算符，只有寄望于不同符号的 nud 方法自己协调好。其实也不难，按照语法规则，不同情况不同处理。

（1）如果刚进入 expression 时的符号是字面量。

执行 advance 之后，当前 token 已经是运算符了，即字面量后面的符号是运算符。那么在该字面量 nud 方法的实现中，就不要去调用 expression，因为 expression 会调用 advance 使 token 更新，所以字面量都是直接返回字面量值。

（2）如果刚进入 expression 时的符号是前缀运算符。

执行 advance 之后，当前 token 是操作数，假设是操作数 *a*，那么该前缀运算符 nud 方法的实现中，就应该递归调用 expression 去获得该前缀运算符的右操作数，也就是消费掉操作数 *a*（前面说过，进入 nud 或 led 方法时，当前 token 就是表达式中方法所属符号的右边的符号），以确保退出 nud 时当前 token 是运算符。

5.3.6　TDOP 总结

因为 TDOP 是用于语法分析，因此不仅是运算符有 led 或 nud 方法，任何具有绑定权值的符号都有自己的 led 或 nud 方法，只是各符号对操作数的绑定权值不同。关注左操作数的符号就有 led 方法。不关注左操作数的符号只有 nud 方法，虽然如此，但这并不表示它就关注右操作数，比如常数就不关注右操作数，它也有 nud 方法，功能是返回自身的值。

led 或 nud 方法中都不会调用 advance，它们只会调用 expression，因为在 led 或 nud 中，当前 token 就是表达式中 led 和 nud 方法所属运算符右边的操作数，而这正是进入 expression 后需要的当前 token。

TDOP 是语法分析的方法，其中的 expression 仅用来计算表达式的结果，并不负责语法分析，语法分析需要不同的语法分析函数专门负责，需要计算表达式时，它们会调用 expression 获得表达式的结果。表达式不限于纯粹的数学表达式，也就是说操作符不仅可以是运算符，还可以是其他语法关键字（一般情况下是与定义有关的语法关键字，比如定义对象直接量），原则上操作符包括 led 或 nud 方法并且操作符有绑定权值，就可以进入 expression 的处理流程。

expression 是由上一个运算符 S 调用的，可以在中置或后置运算符的 led 方法中调用它，也可以在前置运算符的 nud 方法中调用它，都是表示运算符对其右边的操作数的吸引（绑定）能力，所以形参 *rbp* 表示 right bind power，即上一个运算符 S 对其右边的操作数的吸引能力。如果运算符 S 是中置运算符，*rbp* 就是运算符 S 的 l*bp*，因为中置运算符对左右操作数的绑定能力是一样的，l*bp* 是当初通过 symbol(S.id, bp)创建时指定的参数 *bp*，symbol 的实现是使 s.lbp=*bp*。如果运算符 s 是前置运算符，它并不关心左操作数，因此它的 l*bp* 为 0（可见 prefix 的实现）。前置运算符只关注它右边的操作数，因此就要在 nud 方法中调用 expression，这时传给 expression 的 *bp* 就是根据实际的优先级大小设定的。前置运算符的优先级最高，因此往往以最高优先级去调用 expression，如 expression(80)。当然也可以直接调用它，毕竟在 led 和 nud 中也是以权值做参数直接调用



expression。当不经 led 和 nud 而直接调用它时，由于当前并没有运算符，但依然要提供右绑定权值作为参数，按说只有运算符才需要绑定操作数，因此只有运算符才有绑定权值，才能为 expression 提供右绑定权值，也就是说只有运算符才具备调用 expression 的能力。expression（0）一定要理解为是某个 0 绑定权值的运算符在调用它，整个表达式中运算符的绑定权值均比 0 大，因此整个表达式的结果将作为该 0 绑定权值对应运算符的右操作数。

expression 是以“右”绑定权值来调用的，用以找到该绑定权值对应的运算符的“右”操作数。只有运算符右边的数字才需要判定是属于该运算符的右操作数还是属于紧邻数字右边下一个运算符的左操作数，expression（右绑定权值）的使命是为带有右绑定权值的调用者找到“右”操作数。

初始分析表达式时是没有运算符的，即没有调用 expression 的主调方运算符，但一定提供了 *rbp*，未必是 0，时刻记住，expression 的使命是为主调方找到其右操作数，因此初始调用 expression 的情况，主调方相当于为 expression 提供 *rbp* 的“假想运算符”，参数 *rbp* 就是该假想操作符的绑定权值。expression 的参数 *rbp* 决定了右操作数在表达式中的长度。若以整个表达式的结果作为某运算符的右操作数，就应该用最小的绑定权值 0 去调用，即 expression(0)。

对于 expression(0)，以下两种理解是错误的。

（1）千万不要理解为：“用 expression(0)所分析的表达式，原因是左边没有运算符才用 0 去调用它”。这样理解是错误的，因为还可以用非 0 的绑定权值去调用 expression，按照这种理解的话这种非 0 的调用说不通了。

（2）对于 expression(0)千万不要这么理解，由于表达式首端并没有运算符，因此相当于没有对右边操作数的绑定能力，因此通常以 0 绑定力调用它，即 expression(0)，貌似很合理对吧，但如果表达式首端是个前缀运算符比如++呢，如++a+2，依然可以用 expression(0)调用。

对 expression(0)一定要理解为：expression(0)是为了处理完整个表达式。expression 始终是由绑定权值来调用的，而只有运算符才有绑定权值的概念。expression(0)一定是某个 0 绑定权值的运算符在调用它，整个表达式中运算符的绑定权值均比 0 大，因此在 expression 中的 while 循环会处理完整个表达式，相当于整个表达式的结果将作为该 0 绑定权值对应运算符的右操作数。

expression 的参数是右绑定权值，因此通常是由具有右绑定权值的运算符调用的，因为该运算符需要右操作数。expression 的参数右绑定权值的作用是，找出大于此绑定权值的运算符（目的是优先计算优先级更高的运算符），然后用它们的计算结果作为调用 expression 的运算符的右操作数。因此参数“右绑定权值”决定了遍历分析表达式的深度。

第6章　实现语法分析与语义分析

注意，语法分析和语义分析是从上到下进行的，分析过程中会调用很多下层的实现，如果按照从上到下的介绍，要么就会介绍得太浅，要么就从上到下顺着一个分支深入下去，介绍完一个再介绍另外一个子分支，相当于上面只有几根线，而下面是一大堆乱七八糟的东西，不容易理清，而且相互之间还有联系。因此，我们采取自下而上的方式学习，先把底层依赖的部分介绍完，然后再说上面的框架。提醒，语义分析的部分会涉及虚拟机对操作码的处理，这两方面需要配合才能理清原理，之前也说过了为什么不能先介绍虚拟机，因此本章有可能大家会觉得吃力，待虚拟机完成后可再回来重温本章。

6.1 定义指令（本节源码 stepByStep/c6/a）

语法分析是基于指令的理解，我们需要知道指令的意义和行为才能将源码生成对应的指令。也许有读者会问，那你为什么不先讲虚拟机？原因是这样的，首先虚拟机是属于指令执行的部分，属于编译器的后端，按照前后端划分的话也是先进行语法及语义分析，最后再轮到虚拟机，按照此流程顺序似乎较容易接受。另外，虚拟机中也涉及一些前端基础的内容，无论先介绍哪个都会涉及另一方，这两者是相辅相成只有等到虚拟机完成后，两者配合着看才能真正理解，因此本章大家知道语义和语法分析的原理及过程就行了，等虚拟机完成后，才能真正融会贯通。

语法分析的同时做语义分析，也就是要生成代码，我们系统中生成代码指的就是生成指令，这就是我们的目标代码。指令在第 4 章中介绍过了，本节按照之前的描述定义操作码，它在 opcode.inc 中，代码如下。

stepByStep/c6/a/vm/opcode.inc

```
/****************** 有关栈 slot 说明  ******************
1 基于栈的虚拟机，操作数和运算结果都会保存在栈中
2 运行时栈是由 Value 数组模拟的，栈中元素称为 slot
3 有的指令使栈空间增大一些 slot，有的则减小一些 slot
4 函数中的局部变量也要存储在栈中
综上，函数所需要的栈空间需要连同函数内所有操作码对栈空间的需求一并统计
操作码的行为是一定的，对栈的影响也是一定的
因此在定义操作码时，以 slot 为单位，一并描述操作码对栈空间大小的影响
下面以此格式定义操作码及相应影响：
    操作码对栈空间大小的影响(注意，不是操作数)
**********************************************/
OPCODE_SLOTS(LOAD_CONSTANT, 1)
OPCODE_SLOTS(PUSH_NULL, 1)
OPCODE_SLOTS(PUSH_FALSE, 1)
OPCODE_SLOTS(PUSH_TRUE, 1)
```

```
OPCODE_SLOTS(LOAD_LOCAL_VAR, 1)
OPCODE_SLOTS(STORE_LOCAL_VAR, 0)
OPCODE_SLOTS(LOAD_UPVALUE, 1)
OPCODE_SLOTS(STORE_UPVALUE, 0)
OPCODE_SLOTS(LOAD_MODULE_VAR, 1)
OPCODE_SLOTS(STORE_MODULE_VAR, 0)
OPCODE_SLOTS(LOAD_THIS_FIELD, 1)
OPCODE_SLOTS(STORE_THIS_FIELD, 0)
OPCODE_SLOTS(LOAD_FIELD, 0)
OPCODE_SLOTS(STORE_FIELD, -1)
OPCODE_SLOTS(POP, -1)
OPCODE_SLOTS(CALL0, 0)
OPCODE_SLOTS(CALL1, -1)
OPCODE_SLOTS(CALL2, -2)
OPCODE_SLOTS(CALL3, -3)
OPCODE_SLOTS(CALL4, -4)
OPCODE_SLOTS(CALL5, -5)
OPCODE_SLOTS(CALL6, -6)
OPCODE_SLOTS(CALL7, -7)
OPCODE_SLOTS(CALL8, -8)
OPCODE_SLOTS(CALL9, -9)
OPCODE_SLOTS(CALL10, -10)
OPCODE_SLOTS(CALL11, -11)
OPCODE_SLOTS(CALL12, -12)
OPCODE_SLOTS(CALL13, -13)
OPCODE_SLOTS(CALL14, -14)
OPCODE_SLOTS(CALL15, -15)
OPCODE_SLOTS(CALL16, -16)
OPCODE_SLOTS(SUPER0, 0)
OPCODE_SLOTS(SUPER1, -1)
OPCODE_SLOTS(SUPER2, -2)
OPCODE_SLOTS(SUPER3, -3)
OPCODE_SLOTS(SUPER4, -4)
OPCODE_SLOTS(SUPER5, -5)
OPCODE_SLOTS(SUPER6, -6)
OPCODE_SLOTS(SUPER7, -7)
OPCODE_SLOTS(SUPER8, -8)
OPCODE_SLOTS(SUPER9, -9)
OPCODE_SLOTS(SUPER10, -10)
OPCODE_SLOTS(SUPER11, -11)
OPCODE_SLOTS(SUPER12, -12)
OPCODE_SLOTS(SUPER13, -13)
OPCODE_SLOTS(SUPER14, -14)
OPCODE_SLOTS(SUPER15, -15)
OPCODE_SLOTS(SUPER16, -16)
OPCODE_SLOTS(JUMP, 0)
OPCODE_SLOTS(LOOP, 0)
OPCODE_SLOTS(JUMP_IF_FALSE, -1)
OPCODE_SLOTS(AND, -1)
OPCODE_SLOTS(OR, -1)
```

```
66 OPCODE_SLOTS(CLOSE_UPVALUE, -1)
67 OPCODE_SLOTS(RETURN, 0)
68 OPCODE_SLOTS(CREATE_CLOSURE, 1)
69 OPCODE_SLOTS(CONSTRUCT, 0)
70 OPCODE_SLOTS(CREATE_CLASS, -1)
71 OPCODE_SLOTS(INSTANCE_METHOD, -2)
72 OPCODE_SLOTS(STATIC_METHOD, -2)
73 OPCODE_SLOTS(END, 0)
```

本文件是被包含的文件，哪里需要就搬到哪里，OPCODE_SLOTS 是个宏，宏的两个参数是操作码和该操作码对栈的影响，如之前第 4 章中字节码小节所述。此宏由包含此文件的源码定义，见下面 vm.h 中的应用。

stepByStep/c6/a/vm/vm.h

```
…略
 9 //为定义在 opcode.inc 中的操作码加上前缀 OPCODE_
10 #define OPCODE_SLOTS(opcode, effect) OPCODE_##opcode,
11 typedef enum {
12     #include "opcode.inc"
13 } OpCode;
14 #undef OPCODE_SLOTS
15
16 typedef enum vmResult {
17     VM_RESULT_SUCCESS,
18     VM_RESULT_ERROR
19 } VMResult;    //虚拟机执行结果
…略
```

第 10～14 行就是 opcode.inc 的应用，把定义在 opcode.inc 中的操作码加上前缀 OPCODE_，比如 JUMP 会变成 OPCODE_JUMP，程序中用的都是后者。

6.2 核心脚本（本节源码 stepByStep/c6/a）

任何编程语言都会预置核心功能供用户使用，这些核心功能是最基础的构件，所有的用户代码都是基于它们编写的。之前我们在类中绑定了一些方法，它们也是核心代码，用 C 语言实现的，下面是用脚本语言实现的，用脚本语言实现脚本语言的核心，很像自举，这说明为了使用这些基础构件，我们很快就要编译它了，先看定义吧，请见代码 vm/core.script.inc。

stepByStep/c6/a/vm/core.script.inc

```
 1 static const char* coreModuleCode =
 2 "class Null {}\n"
 3 "class Bool {}\n"
 4 "class Num {}\n"
 5 "class Fn {}\n"
 6 "class Thread {}\n"
 7 "\n"
 8 "class Sequence {\n"
 9 "    all(f) {\n"
10 "        var result = true\n"
```

```
"        for element (this) {\n"
"            result = f.call(element)\n"
"            if (!result) return result\n"
"        }\n"
"        return result\n"
"    }\n"
"\n"
"    any(f) {\n"
"        var result = false\n"
"        for element (this) {\n"
"            result = f.call(element)\n"
"            if (result) return result\n"
"        }\n"
"        return result\n"
"    }\n"
"\n"
"    contains(element) {\n"
"        for item (this) if (element == item) return true\n"
"        return false\n"
"    }\n"
"\n"
"    count {\n"
"        var result = 0\n"
"        for element (this) result = result + 1\n"
"        return result\n"
"    }\n"
"\n"
"    count(f) {\n"
"        var result = 0\n"
"        for element (this) if (f.call(element)) result = result + 1\n"
"        return result\n"
"    }\n"
"\n"
"    each(f) {\n"
"        for element (this) f.call(element)\n"
"    }\n"
"\n"
"    isEmpty {\n"
"        return iterate(null) ? false : true\n"
"    }\n"
"\n"
"    map(transformation) {\n"
"        return MapSequence.new(this, transformation)\n"
"    }\n"
"\n"
"    where(predicate) {\n"
"      return WhereSequence.new(this, predicate)\n"
"    }\n"
"\n"
"    reduce(acc, f) {\n"
```

```
"        for element (this) acc = f.call(acc, element)\n"
"        return acc\n"
"    }\n"
"\n"
"    reduce(f) {\n"
"        var iter = iterate(null)\n"
"        if (!iter) Thread.abort(\"Can't reduce an empty sequence.\")\n"
"        var result = iteratorValue(iter)\n"
"        while (iter = iterate(iter)) result = f.call(result, iteratorValue(iter))\n"
"        return result\n"
"  }\n"
"\n"
"   join(sep) {\n"
"        var first = true\n"
"        var result = \"\"\n"
"        for element (this) {\n"
"            if (!first) result = result + sep\n"
"            first = false\n"
"            result = result + element.toString\n"
"        }\n"
"        return result\n"
"   }\n"
"\n"
"   join() {\n"
"        return join(\"\")\n"
"   }\n"
"\n"
"   toList {\n"
"        var result = List.new()\n"
"        for element (this) result.add(element)\n"
"        return result\n"
"   }\n"
"}\n"
"\n"
"class MapSequence < Sequence {\n"
"    var sequence\n"
"    var fn\n"
"    new(seq, f) {\n"
"        sequence = seq\n"
"        fn = f\n"
"    }\n"
"\n"
"   iterate(iterator) { \n"
"        return sequence.iterate(iterator)\n"
"   }\n"
"   iteratorValue(iterator) {\n"
"        return fn.call(sequence.iteratorValue(iterator))\n"
"   }\n"
"}\n"
"\n"
```

```
111 "class WhereSequence < Sequence {\n"
112 "    var sequence\n"
113 "    var fn\n"
114 "    new(seq, f) {\n"
115 "       sequence = seq\n"
116 "       fn = f\n"
117 "    }\n"
118 "\n"
119 "    iterate(iterator) {\n"
120 "        while (iterator = sequence.iterate(iterator)) \n"
121 "          if (fn.call(sequence.iteratorValue(iterator))) break\n"
122 "        return iterator\n"
123 "    }\n"
124 "\n"
125 "    iteratorValue(iterator) {\n"
126 "        return sequence.iteratorValue(iterator)\n"
127 "    }\n"
128 "}\n"
129 "\n"
130 "class String < Sequence {\n"
131 "    bytes { \n"
132 "       return StringByteSequence.new(this)\n"
133 "    }\n"
134 "    codePoints {\n"
135 "       return StringCodePointSequence.new(this)\n"
136 "    }\n"
137 "\n"
138 "    *(count) {\n"
139 "       if (!(count is num) || !count.isInteger || count < 0) \n"
140 "            Thread.abort(\"Count must be a non-negative integer.\")\n"
141 "       var result = \"\"\n"
142 "       for i (0..(count - 1)) result = result + this\n"
143 "       return result\n"
144 "    }\n"
145 "}\n"
146 "\n"
147 "class StringByteSequence < Sequence {\n"
148 "    var string\n"
149 "    new(str) {\n"
150 "       string = str\n"
151 "    }\n"
152 "\n"
153 "    [index] { \n"
154 "       return string.byteAt_(index)\n"
155 "    }\n"
156 "    iterate(iterator) {\n"
157 "       return string.iterateByte_(iterator) \n"
158 "    }\n"
159 "    iteratorValue(iterator) {\n"
160 "        return string.byteAt_(iterator) \n"
```

```
161 "    }\n"
162 "\n"
163 "    count { \n"
164 "       return string.byteCount_ \n"
165 "    }\n"
166 "}\n"
167 "\n"
168 "class StringCodePointSequence < Sequence {\n"
169 "    var string\n"
170 "    new(str) {\n"
171 "       string = str\n"
172 "    }\n"
173 "\n"
174 "    [index] { \n"
175 "       return string.codePointAt_(index)\n"
176 "    }\n"
177 "    iterate(iterator) {\n"
178 "        return string.iterate(iterator) \n"
179 "    }\n"
180 "    iteratorValue(iterator) {\n"
181 "        return string.codePointAt_(iterator)\n"
182 "    }\n"
183 "\n"
184 "    count {\n"
185 "       return string.count \n"
186 "    }\n"
187 "}\n"
188 "\n"
189 "class List < Sequence {\n"
190 "    addAll(other) {\n"
191 "       for element (other) add(element)\n"
192 "       return other\n"
193 "    }\n"
194 "\n"
195 "    toString {\n"
196 "        return \"[%(join(\",\"))]\" \n"
197 "    }\n"
198 "\n"
199 "    +(other) {\n"
200 "       var result = this[0..-1]\n"
201 "       for element (other) result.add(element)\n"
202 "       return result\n"
203 "    }\n"
204 "\n"
205 "    *(count) {\n"
206 "       if (!(count is num) || !count.isInteger || count < 0) \n"
207 "            Thread.abort(\"Count must be a non-negative integer.\")\n"
208 "       var result = []\n"
209 "       for i (0..(count - 1)) result.addAll(this)\n"
210 "       return result\n"
```

```
211 "    }\n"
212 "}\n"
213 "\n"
214 "class Map {\n"
215 "    keys { \n"
216 "        return MapKeySequence.new(this) \n"
217 "    }\n"
218 "    values {\n"
219 "         return MapValueSequence.new(this)\n"
220 "    }\n"
221 "\n"
222 "    toString {\n"
223 "        var first = true\n"
224 "        var result = \"{\"\n"
225 "\n"
226 "        for key (keys) {\n"
227 "            if (!first) result = result + \", \"\n"
228 "            first = false\n"
229 "            result = result + \"%(key): %(this[key])\"\n"
230 "        }\n"
231 "\n"
232 "        return result + \"}\"\n"
233 "    }\n"
234 "}\n"
235 "\n"
236 "class MapKeySequence < Sequence {\n"
237 "    var map\n"
238 "    new(mp) {\n"
239 "        map = mp\n"
240 "    }\n"
241 "\n"
242 "    iterate(n) {\n"
243 "        return map.iterate_(n) \n"
244 "    }\n"
245 "    iteratorValue(iterator) {\n"
246 "        return map.keyIteratorValue_(iterator)\n"
247 "    }\n"
248 "}\n"
249 "\n"
250 "class MapValueSequence < Sequence {\n"
251 "     var map\n"
252 "     new(mp) {\n"
253 "         map = mp\n"
254 "     }\n"
255 "\n"
256 "    iterate(n) {\n"
257 "        return map.iterate_(n) \n"
258 "    }\n"
259 "    iteratorValue(iterator) {\n"
260 "        return map.valueIteratorValue_(iterator) \n"
```

仅供非商业用途或交流学习使用


```
261 "    }\n"
262 "}\n"
263 "\n"
264 "class Range < Sequence {}\n"
265 "\n"
266 "class System {\n"
267 "    static print() {\n"
268 "        writeString_(\"\n\")\n"
269 "    }\n"
270 "\n"
271 "    static print(obj) {\n"
272 "        writeObject_(obj)\n"
273 "        writeString_(\"\n\")\n"
274 "        return obj\n"
275 "    }\n"
276 "\n"
277 "    static printAll(sequence) {\n"
278 "        for object (sequence) writeObject_(object)\n"
279 "        writeString_(\"\n\")\n"
280 "    }\n"
281 "\n"
282 "    static write(obj) {\n"
283 "        writeObject_(obj)\n"
284 "        return obj\n"
285 "    }\n"
286 "\n"
287 "    static writeAll(sequence) {\n"
288 "        for object (sequence) writeObject_(object)\n"
289 "    }\n"
290 "\n"
291 "    static writeObject_(obj) {\n"
292 "        var str = obj.toString\n"
293 "        if (str is String) {\n"
294 "            writeString_(str)\n"
295 "        } else {\n"
296 "            writeString_(\"[invalid toString]\")\n"
297 "        }\n"
298 "    }\n"
299 "}\n";
```

开头定义了一些空的基础类，如 class Thread 等，它们在此仅起到定义的作用，后面我们会往里面绑定方法。后面的 class Sequence 是所有可迭代序列的基础，这是完全定义好的类，大家简单看下其中的方法，此时不必过度关注，待编译器完成后试着应用这些代码时再体会。

6.3 写入指令（本节源码 stepByStep/c6/a）

进入正题之前先看下本节 parser.h 中的更新。

仅供非商业用途或交流学习使用

stepByStep/c6/a/parser/parser.h

```
...略
 94 struct parser {
 95     const char* file;
 96     const char* sourceCode;
 97     const char* nextCharPtr;
 98     char curChar;
 99     Token curToken;
100     Token preToken;
101     ObjModule* curModule;     // 当前正在编译的模块
102     CompileUnit* curCompileUnit;    // 当前编译单元
103
104     //处于内嵌表达式之中时,期望的右括号数量
105     //用于跟踪小括号对儿的嵌套
106     int interpolationExpectRightParenNum;
107     struct parser* parent;  //指向父 parser
108     VM* vm;
109 };
...略
```

本节在之前的基础上多了第 102 行的 curCompileUnit，表示当前编译单元，词法分析器在任意时刻总会处于一个编译单元中，模块、方法、函数都有自己的编译单元。模块也是编译单元的原因是变量可直接定义在模块中，即模块变量（类在模块中也是以模块变量的形式存在），模块变量的存取是有相关指令的，故有独立的指令流。值得注意的是，类没有编译单元，因为编译单元是独立的指令流单位，类是由方法组成的，方法才是指令流的单位，类中不允许定义变量，方法中才可以有变量定义，因此类只是方法的编译单元的集合。

我们的目标代码就是我们自定义的指令，生成的指令包括操作码和操作数两部分，操作码都是一字节大小，操作数有一字节和两字节之分，对于多字节的操作数涉及存储顺序问题，即字节序，我们采取大端字节序列。之前介绍过了，大端字节序就是低地址存储数据的高字节，高地址存储数据的低字节。简单来说，对于十六进制操作数 0x1234，如果用字节数组 *a* 来存储，*a*[0]是 0x12，*a*[1]是 0x34。写入指令的代下面的 compiler.c 中。

stepByStep/c6/a/compiler/compiler.c

```
...略
 41 //把 opcode 定义到数组 opCodeSlotsUsed 中
 42 #define OPCODE_SLOTS(opCode, effect) effect,
 43 static const int opCodeSlotsUsed[] = {
 44     #include "opcode.inc"
 45 };
 46 #undef OPCODE_SLOTS
 47
 48 //初始化 CompileUnit
 49 static void initCompileUnit(Parser* parser, CompileUnit* cu,
 50         CompileUnit* enclosingUnit, bool isMethod) {
 51     parser->curCompileUnit = cu;
 52     cu->curParser = parser;
 53     cu->enclosingUnit = enclosingUnit;
 54     cu->curLoop = NULL;
 55     cu->enclosingClassBK = NULL;
```

仅供非商业用途或交流学习使用


```
56
57      //若没有外层,说明当前属于模块作用域
58      if (enclosingUnit == NULL) {
59          //编译代码时是从上到下从最外层的模块作用域开始,模块作用域设为-1
60          cu->scopeDepth = -1;
61          //模块级作用域中没有局部变量
62          cu->localVarNum = 0;
63
64      } else {    //若是内层单元,属局部作用域
65          if (isMethod) {   //若是类中的方法
66              //如果是类的方法就设定隐式 this 为第 0 个局部变量,即实例对象,
67              //它是方法(消息)的接收者, this 这种特殊对象被处理为局部变量
68              cu->localVars[0].name = "this";
69              cu->localVars[0].length = 4;
70
71          } else {     //若为普通函数
72              //空出第 0 个局部变量,保持统一
73              cu->localVars[0].name = NULL;
74              cu->localVars[0].length = 0;
75          }
76
77          //第 0 个局部变量的特殊性使其作用域为模块级别
78          cu->localVars[0].scopeDepth = -1;
79          cu->localVars[0].isUpvalue = false;
80          cu->localVarNum = 1;  //localVars[0]被分配
81          // 对于函数和方法来说,初始作用域是局部作用域
82          // 0 表示局部作用域的最外层
83          cu->scopeDepth = 0;
84      }
85
86      //局部变量保存在栈中,初始时栈中已使用的 slot 数量等于局部变量的数量
87      cu->stackSlotNum = cu->localVarNum;
88
89       cu->fn = newObjFn(cu->curParser->vm, cu->curParser->curModule, cu->localVarNum);
90  }
91
92  //往函数的指令流中写入 1 字节,返回其索引
93  static int writeByte(CompileUnit* cu, int byte) {
94       //若在调试状态,额外在 debug->lineNo 中写入当前 token 行号
95  #if DEBUG
96      IntBufferAdd(cu->curParser->vm,
97              &cu->fn->debug->lineNo, cu->curParser->preToken.lineNo);
98  #endif
99      ByteBufferAdd(cu->curParser->vm,
100         &cu->fn->instrStream, (uint8_t)byte);
101     return cu->fn->instrStream.count - 1;
102 }
103
104 //写入操作码
105 static void writeOpCode(CompileUnit* cu, OpCode opCode) {
```

仅供非商业用途或交流学习使用


```
106     writeByte(cu, opCode);
107     //累计需要的运行时空间大小
108     cu->stackSlotNum += opCodeSlotsUsed[opCode];
109     if (cu->stackSlotNum > cu->fn->maxStackSlotUsedNum) {
110         cu->fn->maxStackSlotUsedNum = cu->stackSlotNum;
111     }
112 }
113
114 //写入 1 个字节的操作数
115 static int writeByteOperand(CompileUnit* cu, int operand) {
116     return writeByte(cu, operand);
117 }
118
119 //写入 2 个字节的操作数 按大端字节序写入参数,低地址写高位,高地址写低位
120 inline static void writeShortOperand(CompileUnit* cu, int operand) {
121     writeByte(cu, (operand >> 8) & 0xff); //先写高 8 位
122     writeByte(cu, operand & 0xff);        //再写低 8 位
123 }
124
125 //写入操作数为 1 字节大小的指令
126 static int writeOpCodeByteOperand(CompileUnit* cu, OpCode opCode, int operand) {
127     writeOpCode(cu, opCode);
128     return writeByteOperand(cu, operand);
129 }
130
131 //写入操作数为 2 字节大小的指令
132 static void writeOpCodeShortOperand(CompileUnit* cu, OpCode opCode, int operand)
 {
133     writeOpCode(cu, opCode);
134     writeShortOperand(cu, operand);
135 }
…略
```

第 42～46 行是把操作码对栈的影响定义在数组 opCodeSlotsUsed 中，第 42 行是把宏 OPCODE_SLOTS 中的参数 effect 取出来，这样使 opcode.inc 中定义的操作码只剩下操作码使用的栈 slot 数量。这样一来，数组 opCodeSlotsUsed 可以用操作码来索引对应的栈 slot 数量。

第 49 行定义的函数 initCompileUnit 用于初始化编译单元 CompileUnit。第 50～55 行模块、方法、函数都是编译单元 CompileUnit，根据编译单元作用域的不同要分别初始化。第 58 行若 enclosingUnit 为 NULL，说明当前编译的是最外层作用域的编译单元，即模块。模块作用域是顶级作用域，用-1 表示，第 60 行使编译单元的作用域 scopeDepth 为-1。模块中不会存在局部变量，故第 62 行使 localVarNum 为 0。第 64 行是处理非模块作用域，即局部作用域编译单元的初始化。形参 isMethod 是布尔值，为 true 时表示方法，否则表示函数。第 65 行判定编译单元为方法时，就将第 0 个局部变量设置为 this，this 即对象本身，也就是方法的接收者，任何方法中第 0 个局部变量都是 this，这样便于找到对象归属类。否则第 71 行判断为普通函数，为方便统一处理方法和函数，即不用区分是方法还是函数，函数也把第 0 个局部变量空出来置为 NULL，保证局部变量的统一布局。由于方法和函数中第 0 个局部变量比较特殊，第 78 行使其作用域为-1，该变量也并不可能是 upvalue，因为其内层不可能引用上层的 this 变量，this 只暴露给本层对象，故第 79

仅供非商业用途或交流学习使用

行将 isUpvalue 置为 false。第 0 个局部变量已占用了，因此第 80 行将已使用的局部变量个数 localVarNum 置为 1。方法或函数只会在第 0 层局部作用域出现，因此作用域 scopeDepth 初始化为 0。第 87 行使 stackSlotNum 即已使用的 slot 数量等于局部变量的个数。第 89 行新建 objFn 用于存储编译单元的指令流。

第 92 行定义的函数 writeByte 是后面所有指令写入函数的基础，用于在函数的指令流中写入 1 字节数据，返回索引。指令流是字节流，即类型为 ByteBuffer，因此第 99 行调用 ByteBufferAdd 实现数据的添加。如果是调度状态，第 96 行额外写入当前 token 所在的行号，以后可显示指令的源码行。

第 105 行定义的函数 writeOpCode 用于写入操作码，操作码是用 writeByte 写入的。CompileUnit.stackSlotNum 只是辅助统计函数方法等代码单元使用的栈空间，由于入栈出栈重叠，即操作码对栈影响有正有负，stackSlotNum 到最后可能为 0，但这在运行过程中对栈的使用量是不可能为 0 的，因此需要记录栈使用的峰值，这更有说服力，峰值保存在 CompileUnit.fn.maxStackSlotUsedNum，为函数分配运行时栈时将参考此峰值。第 108～111 行在写入操作码的时候，统计编译单元中所有操作码对栈的最终影响，maxStackSlotUsedNum 总是记录最大值，因此是峰值。

第 115 行定义的函数 writeByteOperand 用于写入 1 个字节的操作数，仅是用于写入操作数，与操作码无关。

第 119 行定义的函数 writeShortOperand 用于写入 2 个字节的操作数，同样仅是写入操作数，并且以大端字节序写入。操作数 operand 是 int 型，其大小取决于系统平台，不过肯定是大于 2 字节，第 121 行通过右移 8 位去掉低 8 位，然后与 0xff 做与操作只保留第 8～15 位，这就是操作数的高 8 位，调用 writeByte 将其写入。第 122 行写入低 8 位。

第 126 行 writeOpCodeByteOperand 用于写入操作数为 1 字节大小的指令，第 127 行先写操作码，第 128 行写入操作数。

第 132 行 writeOpCodeShortOperand 函数用于写入操作数为 2 字节大小的指令，第 133 行写入操作码，第 134 行写入 2 字节的操作数。

6.4 编译模块（本节源码 stepByStep/c6/a）

编译模块就是编译文件，这是在 compiler.c 中完成的，相关代码如下。

stepByStep/c6/a/compiler/compiler.c

```
……略
174 //编译程序
175 static void compileProgram(CompileUnit* cu) {
176     ;
177 }
178
179 //编译模块
180 ObjFn* compileModule(VM* vm, ObjModule* objModule, const char* moduleCode) {
181     //各源码模块文件需要单独的parser
182     Parser parser;
183     parser.parent = vm->curParser;
184     vm->curParser = &parser;
185
186     if (objModule->name == NULL) {
```

仅供非商业用途或交流学习使用


```
187         // 核心模块是 core.script.inc
188         initParser(vm, &parser, "core.script.inc", moduleCode, objModule);
189     } else {
190         initParser(vm, &parser,
191                 (const char*)objModule->name->value.start, moduleCode, objModule);
192     }
193
194     CompileUnit moduleCU;
195     initCompileUnit(&parser, &moduleCU, NULL, false);
196
197     //记录现在模块变量的数量,后面检查预定义模块变量时可减少遍历
198     uint32_t moduleVarNumBefor = objModule->moduleVarValue.count;
199
200     //初始的 parser->curToken.type 为 TOKEN_UNKNOWN,下面使其指向第一个合法的 token
201     getNextToken(&parser);
202
203     //此时 compileProgram 为桩函数,并不会读进 token,因此是死循环
204     while (!matchToken(&parser, TOKEN_EOF)) {
205         compileProgram(&moduleCU);
206     }
207
208     //后面还有很多要做的,临时放一句话在这提醒
209     //不过目前上面是死循环,本句无法执行
210     printf("There is something to do...\n"); exit(0);
211 }
```

第 175 行的 compileProgram 函数是编译的入口，目前是空的，下节再往里面装东西。

第 180 行的 compileModule 用于编译模块，形参 objModule 是模块，moduleCode 是模块的源码文件字符串。第 182～184 行创建 parser 并初始化。之前说过，一个词法分析器只负责一个文件，因此 parser 是局部变量创建的以保证唯一性。第 186～192 行是针对核心模块和一般模块的词法分析器初始化工作，区别就是文件名，核心模块就是用脚本源码定义的 core.script.inc 作初始化，否则就用模块名初始化词法分析器。模块也是有编译单元的，因此第 194～195 行创建编译单元并初始化。第 198 行的变量 moduleVarNumBefor 用于记录模块中目前模块变量的数量，后面检查预定义模块变量（即之前所说的以行号为值的预定义模块变量）时可减少遍历。由于初始化时 parser->curToken.type 为 TOKEN_UNKNOWN，这在后面 compileProgram 中会导致停止，因此第 201 行手动调用 getNextToken，以使其指向第一个合法的 token。第 204～206 行的 while 循环是编译的入口，调用 compileProgram 编译程序，直到第 204 行判断遇到了结束 token，即编译完毕为止。后面还有一些代码，以后更新。怕忘了，先在第 210 行打印一句话提示后退出。补充一句，目前 compileProgram 为桩函数，并不会读进 token，因此 while 循环是死循环。

compileModule 是由 loadModule 函数调用的，之前 loadModule 已经放到了函数 executeModule 中，编译核心模块就是执行模块，因此可以将 executeModule 放在 buildCore 中了，代码如下。

stepByStep/c6/a/vm/core.c

```
...略
268 //编译核心模块
269 void buildCore(VM* vm) {
...略
```

仅供非商业用途或交流学习使用


```
310     //执行核心模块
311     executeModule(vm, CORE_MODULE, coreModuleCode);
312 }
...略
```

因为 compileModule 中的死循环，本节代码编译后运行无结果。

6.5 语义分析的本质

语义分析就是把 A 语言转变为 B 语言。通常 A 语言是高级语言，B 语言是低级语言，对于脚本语言来说，语义分析就是把脚本语言转换为指令。语义分析模块起到的是翻译人员的作用，它必须同时懂 A 和 B 两种语言才能把 A 转换为 B，需要从 A 转换为 B 的原因是接收方只懂 B 语言。人们愿意用高级语言编程并不是因为机器语言晦涩难懂，而是因为机器语言烦琐、不方便，因此计算机语言编译的意义是：把用户容易使用的高级语言转换为较繁琐的机器语言。总之并不是人们的智力或能力无法驾驭机器语言，毕竟机器语言也是人发明的，人们只是图省事才用高级语言。而 CPU 只能理解机器语言，不懂高级语言，所以要让 CPU 执行高级语言的意图必须要转换为同等意图的机器语言，这就是语义分析的工作，将 A 语言所表述的意图用 B 语言再表述一次。编译的核心是翻译，即语义分析，前面那堆词法和语法分析都是为了理解源码，理解之后才能翻译为目标代码。通常情况下 A 是宏观语言，B 是微观语言（如机器语言），A 的一句话需要 B 语言中多条语句才能完成。要想实现高级语言的意图，必须要把高级语言转换为 CPU 理解的机器语言，因为只有 CPU 能实现意图。

说说我们的脚本和虚拟机，虚拟机是执行脚本语言意图的实体，我们依赖它使意图成真。虚拟机只认得指令，不认得脚本语言，因此需要在语义分析中把脚本语言转换为它所认识的指令。指令是脚本代码的微操作（微步骤），一句脚本代码的功能需要多条指令来组合完成。我们的语义分析就是把用户代码转换为多条指令，强调一下，各条指令都属于实现的过程，要想实现结果，一个指令都不能少执行，比如脚本代码中加法的操作数是数字常量，我们需要为虚拟机生成加载常量的指令，虚拟机在执行此指令后才能获得该数字常量。大家要清楚，对于脚本语言来说，操作数就摆在代码中，可是虚拟机不懂脚本语言，也就是说虚拟机不懂如何从高级语言中获得操作数，因此我们才需要语义分析为它生成获得操作数的指令。

总结，语义分析就是把脚本代码细化为可以被虚拟机理解的指令的过程，虚拟机执行完这些指令后就实现了脚本代码的意图。理解这一点才能明白语义分析的原理。

6.6 注册编译函数（本节源码 stepByStep/c6/b）

每个符号有自己的语法规则，那么该符号的语法分析也有自己的方法。还记得 TDOP 教程中用 JavaScript 完成的语法分析器吧，其中有函数叫 stmt，专用于为符号注册语法分析函数，本节我们也要实现这个功能。该功能定义在 compiler.c 中，代码如下。

stepByStep/c6/b/compiler/compiler.c

```
...略
48 typedef enum {
49     BP_NONE,       //无绑定能力
50
51     //从上往下,优先级越来越高
```

仅供非商业用途或交流学习使用


```
52      BP_LOWEST,          //最低绑定能力
53      BP_ASSIGN,          // =
54      BP_CONDITION,       // ?:
55      BP_LOGIC_OR,        // ||
56      BP_LOGIC_AND,       // &&
57      BP_EQUAL,           // == !=
58      BP_IS,              // is
59      BP_CMP,             // < > <= >=
60      BP_BIT_OR,          // |
61      BP_BIT_AND,         // &
62      BP_BIT_SHIFT,       // << >>
63      BP_RANGE,           // ..
64      BP_TERM,            // + -
65      BP_FACTOR,          // * / %
66      BP_UNARY,           // - ! ~
67      BP_CALL,            // . () []
68      BP_HIGHEST
69  } BindPower;            //定义了操作符的绑定权值,即优先级
70
71  //指示符函数指针
72  typedef void (*DenotationFn)(CompileUnit* cu, bool canAssign);
73
74  //签名函数指针
75  typedef void (*methodSignatureFn)(CompileUnit* cu, Signature* signature);
76
77  typedef struct {
78      const char* id;             //符号
79
80      //左绑定权值,不关注左边操作数的符号此值为 0
81      BindPower lbp;
82
83      //字面量、变量、前缀运算符等不关注左操作数的 Token 调用的方法
84      DenotationFn nud;
85
86      //中缀运算符等关注左操作数的 Token 调用的方法
87      DenotationFn led;
88
89      //表示本符号在类中被视为一个方法,
90      //为其生成一个方法签名
91      methodSignatureFn methodSign;
92  } SymbolBindRule;    //符号绑定规则
…略
220 //添加常量并返回其索引
221 static uint32_t addConstant(CompileUnit* cu, Value constant) {
222      ValueBufferAdd(cu->curParser->vm, &cu->fn->constants, constant);
223      return cu->fn->constants.count - 1;
224 }
225
226 //生成加载常量的指令
227 static void emitLoadConstant(CompileUnit* cu, Value value) {
```

仅供非商业用途或交流学习使用


```
228      int index = addConstant(cu, value);
229      writeOpCodeShortOperand(cu, OPCODE_LOAD_CONSTANT, index);
230 }
231
232 //数字和字符串.nud() 编译字面量
233 static void literal(CompileUnit* cu, bool canAssign UNUSED) {
234      //literal 是常量(数字和字符串)的 nud 方法,用来返回字面值
235      emitLoadConstant(cu, cu->curParser->preToken.value);
236 }
237
238 //不关注左操作数的符号称为前缀符号
239 //用于如字面量、变量名、前缀符号等非运算符
240 #define PREFIX_SYMBOL(nud) {NULL, BP_NONE, nud, NULL, NULL}
241
242 //前缀运算符，如!
243 #define PREFIX_OPERATOR(id) {id, BP_NONE, unaryOperator, NULL, unaryMethodSignature}
244
245 //关注左操作数的符号称为中缀符号
246 //数组[，函数(，实例与方法之间的.等
247 #define INFIX_SYMBOL(lbp, led) {NULL, lbp, NULL, led, NULL}
248
249 //中缀运算符
250 #define INFIX_OPERATOR(id, lbp) {id, lbp, NULL, infixOperator, infixMethodSignature}
251
252 //既可做前缀又可做中缀的运算符，如-
253 #define MIX_OPERATOR(id) {id, BP_TERM, unaryOperator, infixOperator, mixMethodSignature}
254
255 //占位用的
256 #define UNUSED_RULE {NULL, BP_NONE, NULL, NULL, NULL}
257
258 SymbolBindRule Rules[] = {
259    /* TOKEN_INVALID*/              UNUSED_RULE,
260    /* TOKEN_NUM */                 PREFIX_SYMBOL(literal),
261    /* TOKEN_STRING */              PREFIX_SYMBOL(literal),
262 };
…略
```

第 48～69 行定义的 BindPower 是绑定权值，也就是各个符号的优先级从上到下递增，具体符号见注释。

第 72 行的 DenotationFn 是函数指针，用来统一指向各种符号的 nud 和 led 方法。

方法都有签名，在系统中是用函数的签名做函数的 id，因此为对象寻找方法也是通过方法的签名。如果一个符号是类中的方法的话，就需要为该符号生成签名，第 75 行的 methodSignatureFn 便是签名函数指针，以后会用此指针指向不同的方法。

第 78～92 行的 SymbolBindRule 是符号的绑定规则，类似 JavaScript 中的 original_symbol，对照之前的概念往里套就行了，id 就是符号的字符串，lbp 是左绑定权值，nud 是 nud 方法，led 是 led 方法，methodSign 就是符号的签名函数，这是之前的 JavaScript 没有的，因为那只是实现语法分析器，我们还要实现语义分析，语义分析中涉及方法签名的索引以生成指令。

以下就是一些依赖函数。第 221 行的函数 addConstant 用于添加常量并返回其索引，实现简

单，不说了。

第 227 行的 emitLoadConstant 用于生成加载常量的指令，分两步，先把常量通过 addConstant 添加到常量表，获得常量在常量表中的索引 index，然后调用 writeOpCodeShortOperand 生成操作码为 OPCODE_LOAD_CONSTANT、操作数为 index 的指令。

第 233 行的 literal 是字面量的 nud 方法，字面量即是常量，包括数字和字符串两种。对于字面量的引用，直接生成加载常量的指令即可。比如我们在程序中引用了字面量，如 a=1，数字 1，那么它的语法分析便是由 literal 函数来负责，也就是生成把 1 加载到栈中的指令，这是通过调用函数 emitLoadConstant 实现的：先把数字 1 作为常量添加到常量表，获得其在常量表中的索引，然后再生成指令“OPCODE_LOAD_CONSTANT，index”把索引为 index 的常量从常量表中加载到栈。上一章中说过，进入 led 或 nud 方法中时，当前 token 是字符串中 led 或 nud 所属符号右边的符号，也就是 curToken，那么 preToken 就是 led 或 nud 所属符号。如源码 1+2，执行 1.nul()时，curToken 为+，preToken 为 1。字面量和字符串的值是存储在 token.value 中。

强调，我们是在语法分析和语义分析阶段，并不负责计算表达式，计算表达式结果是运行时阶段的事，编译（语法和语义分析）阶段是在为运行时阶段的虚拟机生成计算表达式的指令，虚拟机执行指令后便完成了表达式的计算，表达式不限于数学表达式，各种语句在 TDOP 中也是表达式。

第 240～256 行定义的宏用于注册符号的字符串 id、绑定权值 lbp、签名函数 methodSign、nud 和 led 方法。宏 PREFIX_SYMBOL 用于注册前缀符号，不包括前缀运算符。宏 PREFIX_OPERATOR 专用于注册前缀运算符。INFIX_SYMBOL 用于注册中缀符号，不包括中缀运算符。INFIX_OPERATOR 专用于注册中缀运算符，MIX_OPERATOR 用于注册同时拥有 led 和 nul 方法的符号。对于没有规则的符号就用 UNUSED_RULE 占位。说明一下，宏中的函数 unaryOperator 和 infixOperator 本节用不到，而且其实现涉及诸多依赖，下节再介绍。

第 258 行定义的数组 Rules 就是用于注册符号的语法和语义分析函数，也就是 led 和 nud 方法集合。这里是用符号的 token 作为索引，也就是说我们用定义在 parser.h 中 Token 的顺序来填充数组 Rules。

注册的框架定了，以后就往数组 Rules 中按照 parser.h 中定义的 Token 顺序添加符号方法。

6.7 赋值运算的条件

我们知道，赋值表达式中等号左边的符号为左值，左值就是“可写”的数据，变量、下标索引和对象字段都是有效的左值。在 JavaScript 版本中的赋值语句会做左值的有效性检查，查看左值是否为变量，或者左值是否为“[”或“.”。但是赋值操作并不仅要求是左值就行了，还需要检查赋值的优先级上下文环境，以确保左值只是赋值符号的左操作数，而不是左值左边运算符的右操作数。也就是说，左值应该只属于赋值运算符的左操作数，这需要赋值运算符的优先级比位于左值左边的运算符优先级更高，这样才能确保左值不被其左边的运算符抢走。比如表 6-1 中的表达式，左列中表达式语法错误，它原本的意思是先把变量 *b* 赋值后再用 *b* 参与其他运算。

表 6-1　　正确的表达式与错误的表达式

假设 *a* 和 *b* 都是变量	
错误	正确
0||*b*=2	0||(*b* =2)
1&&*b* =2	1&&(*b* =2)

续表

假设 *a* 和 *b* 都是变量	
错误	正确
2==*b* =2	2==(*b* =2)
3>*b* =2	3>(*b* =2)
a|*b* =2	*a* |(*b* =2)
a &*b* =2	*a* &(*b* =2)
a <<*b* =2	*a* <<(*b* =2)
a +*b* =2	*a* +(*b* =2)
a **b* =2	*a* *(*b* =2)
~*b* =2	~(*b* =2)

赋值运算符的优先级最低，在表达式中的优先级必然小于它左边的运算符，按照默认的优先级，左列中左值 *b* 沦为左边运算符的右操作数。拿 2==*b*=2 来说，*b* 先成为==的右操作数，其结果无论是 true 还是 false，又会成为=的左值，2 不能赋值给布尔值，这显然是无效的左值。如果想表达“先赋值，再做其他运算”的思路，应该像右列那样加上小括号以提升优先级先计算赋值，即 2==(*b*=2)。提醒一下，也许有读者觉得左列中前两种的逻辑或运算和逻辑与运算没毛病，这是错觉，逻辑运算符的优先级更高，用 0||*b*=2 来说，其逻辑上相当于（0||*b*）=2，数字赋值给布尔值显然是错误的，在 C 语言中也会报错。注意，list[*b*=2]是正确的，不用变成 list[(*b*=2)]，因为“[”识别右操作数是以“]”为边界，中括号中的整体都是“[”的右操作数，所以并不像其他运算符那样右操作数只是识别到了“局部”的变量 *b*，其他语言中也是这样做的。

以上“左列”中赋值运算不成立的原因是=左边的运算符都比=优先级高，左值都被左边的运算符抢走了，按照 expression 中 while 的循环条件，那么只要左边运算符优先级比=低就好了，这个就是赋值运算合法的上下文环境。

至于左值是否有效（是否可写），由各符号自己把握就好，我们是用符号的 led 或 nud 方法做语法分析，那么在各符号的 led 或 nud 方法中再做进一步判断即可。比如变量和列表元素是有效的左值，那么其 nud 方法中就应该考虑到成为左值的情况，也就是检查变量名后面是否有赋值运算符，如=。常量如数字或字符串字面量并不是有效的左值，那么其 nud 方法就不应该考虑成为左值的情况。

另外，尽管这里是用赋值运算符如=举例，但以后你会看到我们并没有为赋值运算符注册语法分析方法，因此并没有如=.led 去分析 *a* = *b*=10 中的 *b*=10。但赋值运算符的环境依然存在，比如在处理 *a*=*b*=10 时，表达式是以从左到右的顺序处理的，我们在 a.nud 方法中可以直接调用 expression(BP_LOWEST)处理其后的 *b*=10，expression 会调用 b.nud，b.nud 方法中同样是用 expression(BP_LOWEST)处理其后的 10。注意 *a* 和 *b* 属于相同的 token，都是变量名，同一类 token 注册的是同样的 nud 方法，因此都是用 expression(BP_LOWEST)处理后面的表达式，我们是在左值的 nud 方法中处理的赋值运算，并没有把赋值运算符单独提出来注册 led 方法。提醒一下，在各个符号的 nud 或 led 方法中并不一定非要用自己的绑定权值调用 expression，这取决于应用环境，如果操作数是表达式，即使在高优先级运算符的 led 方法中也应该用 expression(BP_LOWEST)去获取表达式结果，总之以后的实际应用中会有很多这样的情况。

总结，赋值运算有两个条件：一是左值有效性，二是赋值操作的优先级上下文环境要足够低。下节实践。

6.8 实现 expression 及其周边（本节源码 stepByStep/c6/c）

先介绍个函数，它定义在 core.c 中，代码如下。

stepByStep/c6/c/vm/core.c

```
234 //确保符号已添加到符号表
235 int ensureSymbolExist(VM* vm, SymbolTable* table, const char* symbol, uint32_t length)
 {
236     int symbolIndex = getIndexFromSymbolTable(table, symbol, length);
237     if (symbolIndex == -1) {
238         return addSymbol(vm, table, symbol, length);
239     }
240     return symbolIndex;
241 }
```

第 235 行定义的 ensureSymbolExist 用于确保符号添加到符号表，分两步实现，先通过第 236 行调用函数 getIndexFromSymbolTable 确认符号是否在 table 中，如果不在，则在第 238 行调用 addSymbol 将其添加到 table，最后返回其索引。

下面是 compiler.c 中的一些更新，代码如下。

stepByStep/c6/c/compiler/compiler.c (1/2)

```
…略
94 static uint32_t addConstant(CompileUnit* cu, Value constant);
…略
222 //把 Signature 转换为字符串,返回字符串长度
223 static uint32_t sign2String(Signature* sign, char* buf) {
224     uint32_t pos = 0;
225
226     //复制方法名 xxx
227     memcpy(buf + pos, sign->name, sign->length);
228     pos += sign->length;
229
230     //下面单独处理方法名之后的部分
231     switch (sign->type) {
232         //SIGN_GETTER 形式:xxx，无参数，上面 memcpy 已完成
233         case SIGN_GETTER:
234             break;
235
236         //SIGN_SETTER 形式: xxx=(_),之前已完成 xxx
237         case SIGN_SETTER:
238             buf[pos++] = '=';
239             //下面添加=右边的赋值,只支持一个赋值
240             buf[pos++] = '(';
241             buf[pos++] = '_';
242             buf[pos++] = ')';
243             break;
244
245         //SIGN_METHOD 和 SIGN_CONSTRUCT 形式:xxx(_,...)
246         case SIGN_CONSTRUCT:
```

```
247         case SIGN_METHOD: {
248             buf[pos++] = '(';
249             uint32_t idx = 0;
250             while (idx < sign->argNum) {
251                 buf[pos++] = '_';
252                 buf[pos++] = ',';
253                 idx++;
254             }
255
256             if (idx == 0) { //说明没有参数
257                 buf[pos++] = ')';
258             } else { //用 rightBracket 覆盖最后的,
259                 buf[pos - 1] = ')';
260             }
261             break;
262         }
263
264         //SIGN_SUBSCRIPT 形式:xxx[_,...]
265         case SIGN_SUBSCRIPT: {
266             buf[pos++] = '[';
267             uint32_t idx = 0;
268             while (idx < sign->argNum) {
269                 buf[pos++] = '_';
270                 buf[pos++] = ',';
271                 idx++;
272             }
273             if (idx == 0) { //说明没有参数
274                 buf[pos++] = ']';
275             } else { //用 rightBracket 覆盖最后的,
276                 buf[pos - 1] = ']';
277             }
278             break;
279         }
280
281         //SIGN_SUBSCRIPT_SETTER 形式:xxx[_,...]=(_)
282         case SIGN_SUBSCRIPT_SETTER: {
283             buf[pos++] = '[';
284             uint32_t idx = 0;
285             //argNum 包括了等号右边的一个赋值参数,
286             //这里是在处理等号左边 subscript 中的参数列表,因此减 1.
287             //后面专门添加该参数
288             while (idx < sign->argNum - 1) {
289                 buf[pos++] = '_';
290                 buf[pos++] = ',';
291                 idx++;
292             }
293             if (idx == 0) { //说明没有参数
294                 buf[pos++] = ']';
295             } else { //用 rightBracket 覆盖最后的,
296                 buf[pos - 1] = ']';
```

```
297             }
298
299             //下面为等号右边的参数构造签名部分
300             buf[pos++] = '=';
301             buf[pos++] = '(';
302             buf[pos++] = '_';
303             buf[pos++] = ')';
304         }
305     }
306
307     buf[pos] = '\0';
308     return pos;    //返回签名串的长度
309 }
...略
```

第 94 行添加了函数 addConstant 的声明，有些在其前定义的函数会调用它。

第 223 行的函数 sign2String 用于把签名转换为字符串，并返回其长度。签名中的参数（形参和实参）用_表示，用逗号分隔。代码较简单，不说了。

下面是本节的重点，expression 及一些符号的 nud 和 led 方法的实现。

stepByStep/c6/c/compiler/compiler.c (2/2)

```
...略
355 //语法分析的核心
356 static void expression(CompileUnit* cu, BindPower rbp) {
357     //以中缀运算符表达式 aSwTe 为例，
358     //大写字符表示运算符，小写字符表示操作数
359
360     //进入 expression 时，curToken 是操作数 w， preToken 是运算符 S
361     DenotationFn nud = Rules[cu->curParser->curToken.type].nud;
362
363     //表达式开头的要么是操作数要么是前缀运算符，必然有 nud 方法
364     ASSERT(nud != NULL, "nud is NULL!");
365
366     getNextToken(cu->curParser);  //执行后 curToken 为运算符 T
367
368     bool canAssign = rbp < BP_ASSIGN;
369     nud(cu, canAssign);    //计算操作数 w 的值
370
371     while (rbp < Rules[cu->curParser->curToken.type].lbp) {
372         DenotationFn led = Rules[cu->curParser->curToken.type].led;
373         getNextToken(cu->curParser);  //执行后 curToken 为操作数 e
374         led(cu, canAssign);  //计算运算符 T.led 方法
375     }
376 }
377
378 //通过签名编译方法调用，包括 callX 和 superX 指令
379 static void emitCallBySignature(CompileUnit* cu, Signature* sign, OpCode opcode)
 {
380     char signBuffer[MAX_SIGN_LEN];
381     uint32_t length = sign2String(sign, signBuffer);
```

```
382
383     //确保签名录入到 vm->allMethodNames 中
384     int symbolIndex = ensureSymbolExist(cu->curParser->vm, \
385             &cu->curParser->vm->allMethodNames, signBuffer, length);
386     writeOpCodeShortOperand(cu, opcode + sign->argNum, symbolIndex);
387
388     //此时在常量表中预创建一个空 slot 占位,将来绑定方法时再装入基类
389     if (opcode == OPCODE_SUPER0) {
390        writeShortOperand(cu,  addConstant(cu, VT_TO_VALUE(VT_NULL)));
391     }
392 }
393
394 //生成方法调用的指令,仅限 callX 指令
395 static void emitCall(CompileUnit* cu, int numArgs, const char* name, int length)
 {
396     int symbolIndex = ensureSymbolExist(cu->curParser->vm,
397             &cu->curParser->vm->allMethodNames, name, length);
398     writeOpCodeShortOperand(cu, OPCODE_CALL0 + numArgs, symbolIndex);
399 }
400
401 //中缀运算符.led 方法
402 static void infixOperator(CompileUnit* cu, bool canAssign UNUSED) {
403     SymbolBindRule* rule = &Rules[cu->curParser->preToken.type];
404
405     //中缀运算符对左右操作数的绑定权值一样
406     BindPower rbp = rule->lbp;
407     expression(cu, rbp);  //解析右操作数
408
409     //生成一个参数的签名
410     Signature sign = {SIGN_METHOD, rule->id, strlen(rule->id), 1};
411     emitCallBySignature(cu, &sign, OPCODE_CALL0);
412 }
413
414 //前缀运算符.nud 方法, 如-、!等
415 static void unaryOperator(CompileUnit* cu, bool canAssign UNUSED) {
416     SymbolBindRule* rule = &Rules[cu->curParser->preToken.type];
417
418     //BP_UNARY 作为 rbp 去调用 expression 解析右操作数
419     expression(cu, BP_UNARY);
420
421     //生成调用前缀运算符的指令
422     //0 个参数,前缀运算符都是 1 个字符,长度是 1
423     emitCall(cu, 0, rule->id, 1);
424 }
…略
```

第 356 行定义的 expression 就是那个 JavaScript 版本 expression 的 C 版本，因此思路是一样的，这里的介绍也只是复习。强调一下，expression 函数并不用于语法分析，语法分析是 TDOP 中的 TD 两个字负责的，即自上而下调用不同的语法分析函数进行语法分析，expression 只负责计算表达式的结果，表达式包括数学表达式和其他操作符组成的表达式，比如“{”是用于 map 字面量

定义操作符，“[”是用于列表字面量定义的操作符。语法分析函数中需要计算表达式的结果时就调用 expression，总之，expression 只是 TDOP 中的 OP 部分，并不用于语法分析，仅用来获取表达式结果。

代码中依然是以中缀运算符表达式 aSwTe 为例辅助介绍。但大家一定要清楚，此处用中缀运算符只是因为较容易介绍 expression，对于前缀运算符，操作数 w 就不适合了（如果是前缀运算符，则操作数 w 所在位置的字符就是前缀运算符了）。

第 361 行是从 Rules 数组获取 nud 方法，这是提前注册好的，因此第 364 行通过 ASSERT 保证 nud 方法一定存在。

第 366 行调用 getNextToken 使 curToken 为运算符 T。

第 368 行的代码 bool canAssign = rbp < BP_ASSIGN 用于判断是否具备可赋值的环境，即当 rbp 小于 BP_ASSIGN 时才保证左值属于赋值运算符，前面花了一节讨论它，忘了的读者回去再看看。

第 369 行调用 nud 方法计算操作数 w 的值。

第 371 行的 while 循环条件大家很熟悉，理想情况下 curToken 在 nud 方法执行后依然为运算符 T，也就是操作数 w 是字面量和简单变量等情况，不过不重要，nud 方法执行后一定会使 curToken 变成运算符。第 372 行获得 curToken 的 led 方法。第 373 行调用 getNextToken 使 curToken 为操作数 e，操作数是为运算符 T.led 方法准备的。第 374 行调用运算符 T.led 方法。

第 378 行的函数 emitCallBySignature 用于通过签名生成方法调用的指令，方法调用包括一般方法和基类方法，也就是参数 opcode 必为 OPCODE_CALL0 或 OPCODE_SUPER0，后面会根据签名 sign 中的方法参数个数 x 确定最终的操作码 OPCODE_CALLx 或 OPCODE_SUPERx。第 381 行调用 sign2String 把签名转换为字符串存储到 signBuffer 中，返回值是签名的长度，存储到变量 length 中。第 384 行确保方法签名在 vm->allMethodNames 中。第 386 行根据签名中的参数个数 argNum 写入合适的指令 OPCODE_CALLargNum 或 OPCODE_SUPER argNum，当然 argNum 小于等于 16，因为操作码范围就是 OPCODE_CALL0～16 和 OPCODE_SUPER0～16。第 389 行判断若是基类方法调用的话，第 390 行先写入 VT_NULL 为基类预留一个空位，因为不能保证此时基类已经被编译完成了，基类可以是在子类后面定义的，此时可能尚不知道基类信息，所以将来编译完成后再装入基类。

第 395 行定义的函数 emitCall 用于生成方法调用的指令，但仅限于 OPCODE_CALLx 系列，参数 name 是方法名，length 是方法名长度。第 396 行确保方法 name 已经录入到 vm->allMethod Names，获得 name 的索引 symbolIndex，第 398 行生成调用该方法的指令。

第 402 行定义的函数 infixOperator 是所有中缀运算符的 led 方法，也就是用此方法为中缀运算符做语法分析，宏 INFIX_OPERATOR 中引用的就是此方法。比如表达式 3+2 会被视为 3.+(2)，数字 3 是对象，+是方法，2 是方法的参数。切记，进入任何一个符号的 led 或 nud 方法时，preToken 是该方法所属的符号（即操作符），curToken 是该方法所属符号右边的符号（即操作数），第 403 行获得该中缀运算符的规则。第 406～407 行利用规则中的 lbp 作为 rbp 调用 expression 去解析右操作数，因为中缀运算符对左右操作数的绑定权值一样。第 388～389 行生成一个参数的方法签名，方法形如 op1.operator(op2)，只有一个参数即 op2。

第 415 行定义的 unaryOperator 是所有前缀运算符的 nud 方法，比如在分析代码!3 时会调用此方法，这些前缀运算符都是单字符，在宏 PREFIX_OPERATOR 中引用的是引方法。先获得该前缀运算符的规则 rule，此时不能用 rule->lbp 去调用 expression，因为前缀运算符不关注左边操作数，所以在 rules 数组中已经设置其 lbp 为 BP_NONE，前缀运算符只关注其右操作数，因此它需要 rbp，但 SymbolBindRule 中没有 rbp 属性，上一章中也介绍过了，对于前缀运算符，由于没有 rbp 就自由设定一个很高的绑定权值，故这里用 BP_UNARY 作为它的 rbp 去调用 expression 分析右操作数。最后生成调用单字符前缀运算符的指令，0 个参数，单字符前缀运算符的长度是 1。也

许你对 0 个参数感到奇怪，代码!3 按照“对象.方法”来说，逻辑上会被处理为 3.!，你看方法！没有参数。其实即使是参数为 0，在虚拟机中也会默认有个隐藏参数，就是对象实例，即消息的接收者，在此处就是 3，3 被按照数字对象来处理，因此虚拟机会使参数个数加上 1。

下节见。

6.9 局部变量作用域管理

作用域大家都清楚，就是变量的可见范围，在某作用域内定义的变量只能被该作用域及其内层作用域访问，对其外层作用域不可见。内层作用域中定义的变量如果与外层作用域同名，就会屏蔽外层变量。如果内层作用域中不存在引用的变量，那么会引用与其最近的外层作用域中的变量，如图 6-1 所示。

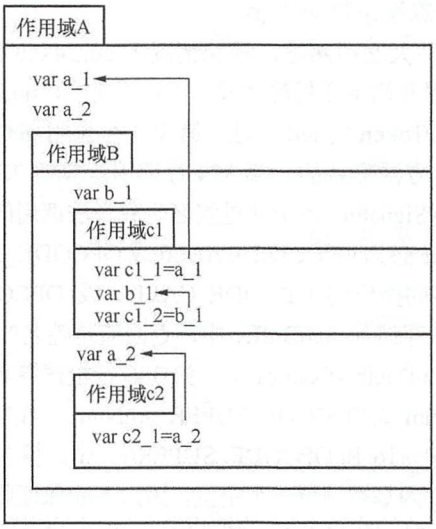


图 6-1

作用域按照深度从-1 开始递增编号，模块作用域是顶级作用域，其作用域编号为-1，并且只有模块作用域的编号是-1。位于模块作用域中的变量称为模块变量（模块变量存储在模块结构 ObjModule 中的 globalVarName 和 globalVarValue 中），除了最顶级的模块作用域外，任何作用域都是局部作用域，-1 已经专用于模块作用域了，因此局部作用域按照深度从 0 递增编号，数字越大说明作用域越深，作用域深度肯定是由于作用域嵌套导致的，比如图 6-1 中作用域 B 嵌套了作用域 C1 和 C2，作用域 C1 的编号就比作用域 B 大 1。值得注意的是，同一深度的作用域在代码中直观上像是同时存在的，但在编译管理时是不可能同时存在的，任何时刻编译器都处于某个作用域中去编译该作用域中的代码，这个作用域就叫当前作用域，比如编译器进到某个作用域（比如作用域 C1）去编译该作用域中的代码，编译完后需要退出该作用域回到其父作用域（比如作用域 B）继续编译，然后又碰到作用域 C2，随后进入 C2 中编译作用域 C2 中的代码，完成后又回到作用域 B。你看，当前作用域的变化轨迹是“B->C1->B->C2->B”。任意时刻编译器只能位于它们中的一个之内。作用域 C1 和 C2 是同级的，但由于不同时存在，因此它们的作用域编号是相同的，并不会导致冲突。以后在实现中你会了解，进入一个作用域时我们会用“CompileUnit->scopdeDepth++”，退出作用域时会用“CompileUnit->scopdeDepth--”，假设作用域 B 的编号是 6，那么作用域 C1 和 C2 的编号都会是 7。我们对作用域深度并没有限制，这完全是由存储编号的变量 scopeDepth 的宽度决定的，它是 int 型，具体大小取决于平台。我们对编译单元 CompileUnit 中的

局部变量总数有限制，最大 128 个，注意，不是同一作用域中允许 128 个局部变量，而是编译单元中所有局部作用域的局部变量总数不能超过 128，比如一个函数或方法中最多可定义 128 个局部变量。下面是局部变量的管理策略。

尽管不同深度的局部作用域都可以有自己的局部变量，但我们并不是以作用域为单位管理局部变量，而是把局部变量的信息（不包括值）统一汇总在编译单元 CompileUnit 中的 localVars 数组中。不过大家不要觉得这很混乱，所有局部变量是按作用域深度排在数组 localVars 中的，并不会出现作用域深度混杂排列的情况，比如这种情况是不会出现的——“局部变量 localVars[3]的 scopeDepth 是 7，localVars[4]的 scopeDepth 是 4，localVar[5]的 scopeDepth 是 9”，因为我们在退出局部作用域时会把该局部作用域中的局部变量从数组 localVars 中“删除”，数组长度是用成员 localVarNum 记数的，每添加一个局部变量就使 localVarNum++，因此 localVarNum -1 始终是最深作用域中最后一个局部变量的索引，添加局部变量肯定是发生在进入该局部作用域时发生的，因此 localVarNum -1 也是当前作用域中最后一个局部变量的索引。退出局部作用域时“删除”变量的原理同样很简单，就是使 localVarNum-=“该局部作用域中局部变量的个数”，退出作用域就是回到父作用域，也就是当前作用域由内层局部作用域变成了该内层局部作用域的直接外层作用域，总之，除了 CompileUnit->localVars [0]已经在 initCompileUnit 函数中被用于存储 this 对象（方法的 CompileUnit）或填充为 null（函数的 CompileUnit）外，数组中其他局部变量是以作用域的深度递增排列，当前局部作用域（同样是最深作用域）中最后一个局部变量始终是 CompileUnit->localVars [CompileUnit->localVarNum-1]。

下一节讲述如何实践。

6.10 变量声明、中缀、前缀及混合运算符方法签名（本节源码 stepByStep/c6/d）

往 rules 数组中注册方法是通过一些宏（比如 PREFIX_OPERATOR）完成的，这些宏中还引用了方法的签名函数，用此签名函数生成相应符号的签名。别忘记了签名的作用，对象在调用一个方法时，系统是以方法的签名去检索方法体，签名是方法的身份证。部分签名函数依赖于变量声明的功能，先完成这些依赖，代码如下。

stepByStep/c6/d/compiler/compiler.c (1/2)

```
...略
//添加局部变量到 cu
static uint32_t addLocalVar(CompileUnit* cu, const char* name, uint32_t length) {
    LocalVar* var = &(cu->localVars[cu->localVarNum]);
    var->name = name;
    var->length = length;
    var->scopeDepth = cu->scopeDepth;
    var->isUpvalue = false;
    return cu->localVarNum++;
}

//声明局部变量
static int declareLocalVar(CompileUnit* cu, const char* name, uint32_t length) {
    if (cu->localVarNum >= MAX_LOCAL_VAR_NUM) {
        COMPILE_ERROR(cu->curParser, "the max length of local variable of one
```

```
            scope is %d", MAX_LOCAL_VAR_NUM);
325     }
326
327     //判断当前作用域中该变量是否已存在
328     int idx = (int)cu->localVarNum - 1;
329     while (idx >= 0) {
330         LocalVar* var = &cu->localVars[idx];
331
332         //只在当前作用域中查找同名变量,
333         //如果到了父作用域就退出,减少没必要的遍历
334         if (var->scopeDepth < cu->scopeDepth) {
335             break;
336         }
337
338         if (var->length == length && memcmp(var->name, name, length) == 0) {
339             char id[MAX_ID_LEN] = {'\0'};
340             memcpy(id, name, length);
341                     COMPILE_ERROR(cu->curParser, "identifier \"%s\" redefinition!
              ", id);
342         }
343         idx--;
344     }
345
346     //检查过后声明该局部变量
347     return addLocalVar(cu, name, length);
348 }
349
350 //根据作用域声明变量
351 static int declareVariable(CompileUnit* cu, const char* name, uint32_t length) {
352     //若当前是模块作用域就声明为模块变量
353     if (cu->scopeDepth == -1) {
354         int index = defineModuleVar(cu->curParser->vm,
355                     cu->curParser->curModule,name,length,VT_TO_VALUE(VT_NULL));
356         if (index == -1) {    //重复定义则报错
357             char id[MAX_ID_LEN] = {'\0'};
358             memcpy(id, name, length);
359              COMPILE_ERROR(cu->curParser,"identifier\"%s\"redefinition!", id);
360         }
361         return index;
362     }
363
364     //否则是局部作用域,声明局部变量
365     return declareLocalVar(cu, name, length);
366 }
367
```

第 312 行定义的函数 addLocalVar 用于往编译单元 cu 中添加局部变量并返回该变量的索引。局部变量存储在 cu->localVars 数组中，cu->localVarNum 是局部变量的总数，由于数组下标是从 0 开始，因此 cu->localVarNum 也是数组中下一个空闲局部变量的索引。下面是把局部变量的属性对号入座，说一下第 316 行代码，局部变量的作用域 var->scopeDepth 就是它所属的作用域，也就

是它所在作用域的编号，该编号记录在编译单元的 scopdeDepth 中，编译器每进入一个作用域就同步更新 CompileUnit->scopeDepth。第 317 行默认使 isUpvalue 为 false，当该局部变量被其内层作用域引用时，由其内层作用域中的使用者将此 isUpvalue 置为 true。添加局部变量后第 318 行使 localVarNum++并返回，即返回新添加变量的索引。

第 322 行的 declareLocalVar 函数用于声明名为 name、长度为 length 的局部变量。第 323 行判断是否超过最大数量。第 328～344 行判断局部变量 name 是否已在当前作用域中存在。scopeDepth 越大表示作用域越深（越局部），上节中说过 localVars 数组中的局部变量是按照作用域深度递增排序的，这里是从后往前找，逆序查找肯定是从当前作用域开始。变量同名冲突检测只需要在当前作用域中查找，第 334 行发现如果到了父作用域就退出遍历，减少没必要的遍历。由于是从后往前找，最后的局部变量永远是当前作用域中的变量，而且到了父作用域时会退出，因此第 338 行只需要判断名字及长度是否匹配就行了，不用额外判断 var->scopeDepth==cu->scopeDepth。检查过后第 346 行调用 addLocalVar 函数添加该局部变量完成声明。

第 351 行定义的 declareVariable 函数会根据作用域声明变量。当前作用域是记录在 Compile Unit->scopeDepth 中，第 353 行若发现其为-1，说明当前位于模块作用域中，那么肯定是定义模块变量，下面调用 defineModuleVar 定义该模块变量，若重定义则报错。否则是位于局部作用域，也就是声明局部变量，第 365 行调用 declareLocalVar 声明局部变量并返回局部变量的索引。

stepByStep/c6/d/compiler/compiler.c (2/2)

```
368 //为单运算符方法创建签名
369 static void unaryMethodSignature(CompileUnit* cu UNUSED, Signature* sign UNUSED) {
370     //名称部分在调用前已经完成,只修改类型
371     sign->type = SIGN_GETTER;
372 }
373
374 //为中缀运算符创建签名
375 static void infixMethodSignature(CompileUnit* cu, Signature* sign) {
376     //在类中的运算符都是方法,类型为 SIGN_METHOD
377     sign->type = SIGN_METHOD;
378
379     // 中缀运算符只有一个参数,故初始为 1
380     sign->argNum = 1;
381     consumeCurToken(cu->curParser, TOKEN_LEFT_PAREN, "expect '(' after infix
        operator!");
382     consumeCurToken(cu->curParser, TOKEN_ID, "expect variable name!");
383                 declareVariable(cu,cu->curParser->preToken.start,cu->curParser
                    ->preToken.length);
384      consumeCurToken(cu->curParser, TOKEN_RIGHT_PAREN, "expect ')' after parameter!");
385 }
386
387 //为既做单运算符又做中缀运算符的符号方法创建签名
388 static void mixMethodSignature(CompileUnit* cu, Signature* sign) {
389     //假设是单运算符方法,因此默认为 getter
390     sign->type = SIGN_GETTER;
391
392     //若后面有'(',说明其为中缀运算符,那就置其类型为 SIGN_METHOD
```

```
393     if (matchToken(cu->curParser, TOKEN_LEFT_PAREN)) {
394         sign->type = SIGN_METHOD;
395         sign->argNum = 1;
396             consumeCurToken(cu->curParser, TOKEN_ID, "expect variable name!");
397                     declareVariable(cu, cu->curParser->preToken.start, cu->
                        curParser->preToken.length);
398         consumeCurToken(cu->curParser, TOKEN_RIGHT_PAREN, "expect ')' after
            parameter!");
399     }
400 }
…略
```

如果签名函数不提前准备好，后面的内容没法介绍，可是不介绍后面内容的话，大家又可能不了解签名函数的逻辑，理解起来会有困难。

有些内容要提前灌输给读者。对象调用方法时，系统也是根据方法的签名从 VM->allMethodNames 中检索方法的索引，然后用此索引直接在对象所属类中找方法体。什么时候为方法创建签名呢？最直接省事的做法是编译类的方法时顺便为方法生成签名，签名是签名函数生成的，因此签名函数肯定是由负责编译方法的函数（后面会知道它是 compileMethod）调用的。如果符号是类中的方法，也就是符号是方法名，那么该符号肯定在数组 Rules 中已经注册了该符号的方法签名函数。同一种类型的符号用的签名函数肯定是一样的，因此只要 compileMethod 用符号的类型即 token.type（后面会知道是 curToken.type），就可以从数组 Rules 中检索到相应的方法签名函数。

方法有多种类型，有 getter、setter、method、subscript 和 subscriptgetter，这几类方法中，方法名是共同的部分，差别在于方法名后面的部分，因此在调用签名函数前我们会先调用 getNextToken 使 curToken（原本是方法名）变成方法名右边的符号，然后调用各方法的签名函数，各签名函数只需要处理自己不同的部分即可。

以上说的有的是未来的内容，以后大家看相关代码时就了解了，现在大家只要了解“进入签名函数时，preToken 是方法名，curToken 是方法名右边的符号”就可以了，比如对于方法 test(a) 来说，test 对应的签名函数中，preToken 是方法名 test 对应的 TOKEN_ID，curToken 是(对应的 TOKEN_LEFT_PAREN，下面回来说签名函数。

第 369 行的 unaryMethodSignature 函数为单运算符方法创建签名，此函数在被调用前已经完成签名了，因为单运算符的方法名和 getter 是一样的，在主调方（compileMethod）已经获得了其方法名，现在只修改类型为 SIGN_GETTER，以后实现编译 class 时就清楚了。

第 375 行定义的函数 infixMethodSignature 为中缀运算符创建签名。在类中的运算符都是方法，也就是说其主调方（compileMethod）已经获得的方法名就是中缀运算符。中缀运算符有两个操作数，左操作数为对象，如 2+3 被编译为 2.+(3)，左操作数是数字 2，数字 2 被按照对象处理。中缀运算符的右操作数就是方法（即中缀运算符）的参数，有参数的方法类型是 method，因此第 377 行置 sign->type 为 SIGN_METHOD。中缀运算符只有一个参数，如 op1.+(op2)中只有参数 op2，所以第 380 行将参数 sign->argNum 置为 1。后面继续读取参数的左小括号，形参名，然后第 383 行把形参声明为变量，最后再读取形参后面的右小括号。

第 388 行的 mixMethodSignature 为既做前缀运算符又做中缀运算符的符号方法创建签名。第 390 行默认为前缀运算符，即相当于 getter，因此类型为 SIGN_GETTER。第 393 行判断，若后面有“(”，说明其需要参数，也就是属于需要两个操作数的中缀运算符，这是普通方法 method，那就置其类型为 SIGN_METHOD，然后后面按照 method 类型来识别，声明形参。

6.11 解析标识符（本节源码 stepByStep/c6/e）

本节实现标识符的解析，标识符是用来标识数据的符号，也就是数据的名字，如变量名、函数名、方法名、类名等。本节内容有点多，需要拆成多个部分。

6.11.1 处理参数列表及相关

参数包括实参和形参两种，形参是函数中的局部变量，实参是赋给形参的值，处理方式自然不同。还有一些基础结构要提前实现，代码如下。

stepByStep/c6/e/compiler/compiler.c(1/7)

```
…略
typedef enum {
    VAR_SCOPE_INVALID,
    VAR_SCOPE_LOCAL,      //局部变量
    VAR_SCOPE_UPVALUE,   //upvalue
    VAR_SCOPE_MODULE    //模块变量
} VarScopeType;    //标识变量作用域

typedef struct {
    VarScopeType scopeType;    //变量的作用域
    //根据 scodeType 的值,
    //此索引可能指向局部变量或 upvalue 或模块变量
    int index;
} Variable;
…略
static uint32_t addConstant(CompileUnit* cu, Value constant);
static void expression(CompileUnit* cu, BindPower rbp);
static void compileProgram(CompileUnit* cu);
…略
//声明模块变量,与 defineModuleVar 的区别是不做重定义检查,默认为声明
static int declareModuleVar(VM* vm, ObjModule* objModule,
                            const char* name, uint32_t length, Value value) {
    ValueBufferAdd(vm, &objModule->moduleVarValue, value);
    return addSymbol(vm, &objModule->moduleVarName, name, length);
}

//返回包含 cu->enclosingClassBK 的最近的 CompileUnit
static CompileUnit* getEnclosingClassBKUnit(CompileUnit* cu) {
    while (cu != NULL) {
        if (cu->enclosingClassBK != NULL) {
            return cu;
        }
        cu = cu->enclosingUnit;
    }
    return NULL;
}

//返回包含 cu 最近的 ClassBookKeep
```

```
419 static ClassBookKeep* getEnclosingClassBK(CompileUnit* cu) {
420     CompileUnit* ncu = getEnclosingClassBKUnit(cu);
421     if (ncu != NULL) {
422         return ncu->enclosingClassBK;
423     }
424     return NULL;
425 }
426
427 //为实参列表中的各个实参生成加载实参的指令
428 static void processArgList(CompileUnit* cu, Signature* sign) {
429     //由主调方保证参数不空
430     ASSERT(cu->curParser->curToken.type != TOKEN_RIGHT_PAREN &&
431            cu->curParser->curToken.type != TOKEN_RIGHT_BRACKET, "empty
               argument list!");
432     do {
433        if (++sign->argNum > MAX_ARG_NUM) {
434            COMPILE_ERROR(cu->curParser, "the max number of argument is %d!",
               MAX_ARG_NUM);
435        }
436        expression(cu, BP_LOWEST);  //加载实参
437     } while (matchToken(cu->curParser, TOKEN_COMMA));
438 }
439
440 //声明形参列表中的各个形参
441 static void processParaList(CompileUnit* cu, Signature* sign) {
442     ASSERT(cu->curParser->curToken.type != TOKEN_RIGHT_PAREN &&
443            cu->curParser->curToken.type != TOKEN_RIGHT_BRACKET, "empty
               argume-nt list!");
444     do {
445        if (++sign->argNum > MAX_ARG_NUM) {
446            COMPILE_ERROR(cu->curParser, "the max number of argument is %d!",
               MAX_ARG_NUM);
447        }
448            consumeCurToken(cu->curParser, TOKEN_ID, "expect variable name!");
449                    declareVariable(cu, cu->curParser->preToken.start, cu->
                       curParser->preToken.length);
450     } while (matchToken(cu->curParser, TOKEN_COMMA));
451 }
```

第 41～46 行定义的 VarScopeType 是按照作用域划分的变量类型。

第 48～53 行定义的 Variable 是变量类型，只用于内部的变量查找之用，并不是具体的变量。

第 400 行定义的 declareModuleVar 用于声明模块变量。

我们是以实践为主，最好是自下而上地介绍，不过很无奈的是，很多前面依赖的函数需要后面的知识才能完全理解，可如果要是先介绍后面的，这就成了自上而下了，前面这些依赖就是黑盒子，自上而下是种“简介”似的学习，对于求知欲很强的人还是觉得不够的。为了大家容易理解，我在前面尽量多介绍，将来实现后面的时候你就会有恍然大悟的感觉。当然，希望将来你把所有代码跟一遍，那样才能融会贯通。

类没有编译单元，但相当于模块中的模块变量，模块有编译单元，当编译一个类时，为了显

示正在编译一个类，我们把模块编译单元中的 enclosingClassBK 指向这个类的 ClassBookKeep 结构。方法也有自己的编译单元，用其编译单元结构中的 enclosingUnit（相当于父编译单元指针、直接外层编译单元指针）指向模块的编译单元（并不指向类的编译单元，因为类没有编译单元，类属于模块中的模块变量）以表示嵌套的层次关系。反过来说，如果当前编译单元的 enclosingUnit 指向的父编译单元中的 ClassBookKeep 并不为空（不空就是指向类编译信息），表示当前编译的是类的方法而不是函数或其他编译单元。

编译单元是可以多层嵌套的，第 408 行定义的函数 getEnclosingClassBKUnit 返回包含 cu->enclosingClassBK 的最近的 CompileUnit，实现比较简单，循环调用 enclosingUnit，直到找到满足 cu->enclosingClassBK != NULL 为止，返回编译单元 cu，注意并不是返回 enclosingClassBK。单看此函数似乎并没什么用，它是由下面的 getEnclosingClassBK 函数调用的。

很多时候我们想获得方法所属的类，也就是想获得包含方法编译单元的父编译单元中的 enclosingClassBK 指向的 ClassBookKeep 结构。第 419 行定义的函数 getEnclosingClassBK 用于满足此需求，它返回包含 cu 最近的 ClassBookKeep。它是调用上面的 getEnclosingClassBKUnit 获得编译单元 ncu，若 ncu 不为空则返回其中的 enclosingClassBK，否则返回 null。

实参的英文单词是 argument，第 428 行定义的 processArgList 函数用于为实参列表中的各个实参生成加载实参的指令。函数调用时的实参是以逗号为分隔的列表。第 432～437 行以逗号为分隔，循环调用 expression 计算每个实参，其实就是生成加载实参到栈中的指令。这里处理实参的顺序是自左到右处理实参列表，按照自左向右的顺序压入实参，被调方法或函数会从栈中获取参数。切记，expression 就是用来计算表达式的，表达式可以是变量、常量、函数调用或数学表达式，总之需要计算表达式（严谨地说是生成计算表达式的一系列指令，虚拟机在执行过这些指令后就计算出了表达式的结果）的地方就要调用 expression。

形参的英文单词是 parameter，第 441 行定义的 processParaList 函数用于声明形参列表中的各个形参。形参是函数或方法中的局部变量，因此在处理函数或方法定义时需要把这些形参按照局部变量的形式声明。形参也是用逗号分隔的参数列表。第 444～450 行以逗号为分隔，循环调用 declareVariable 声明各个形参。

这只是本次更新的第一部分，其他部分见后面小节。

6.11.2 实现运算符和标识符的签名函数

系统是依靠方法的签名来检索方法，签名是方法声明的字符串形式，即方法原型的字符串化。创建签名是在编译方法时完成的，换句话说，如果某个符号（任意符号，如运算符、标识符等）作为类中的方法（也就是该符号为方法名），那么该符号一定要有匹配的签名函数为其创造签名。为什么要调用与符号匹配的签名函数？难道不能通用吗？答案是肯定不能通用，签名中是要包含形参列表的，也就是操作数个数，比如中缀运算符有两个操作数，而前缀运算符只有一个操作数，操作数个数不同签名也必然不同。签名函数是注册到数组 Rules 中的，由需要它的函数通过符号对应的 token 到该数组中匹配。

在以后的一段时间，我们都是在准备这些基础环境，本节先完成一小部分，代码如下。

stepByStep/c6/e/compiler/compiler.c(2/7)

```
452
453 //尝试编译 setter
454 static bool trySetter(CompileUnit* cu, Signature* sign) {
455     if (!matchToken(cu->curParser, TOKEN_ASSIGN)) {
456         return false;
```

```
457     }
458
459     if (sign->type == SIGN_SUBSCRIPT)  {
460         sign->type = SIGN_SUBSCRIPT_SETTER;
461     } else {
462         sign->type = SIGN_SETTER;
463     }
464
465     //读取等号右边的形参左边的(
466     consumeCurToken(cu->curParser, TOKEN_LEFT_PAREN, "expect '(' after '='!");
467
468     //读取形参
469     consumeCurToken(cu->curParser, TOKEN_ID, "expect ID!");
470     //声明形参
471             declareVariable(cu, cu->curParser->preToken.start, cu->curParser
                ->preToken.length);
472
473     //读取等号右边的形参右边的(
474     consumeCurToken(cu->curParser, TOKEN_RIGHT_PAREN, "expect ')' after
        argument list!");
475     sign->argNum++;
476     return true;
477 }
478
479 //标识符的签名函数
480 static void idMethodSignature(CompileUnit* cu, Signature* sign) {
481
482     sign->type = SIGN_GETTER;  //刚识别到 id,默认为 getter
483
484     //new 方法为构造函数
485     if (sign->length == 3 && memcmp(sign->name, "new", 3) == 0) {
486
487         //构造函数后面不能接=,即不能成为 setter
488         if (matchToken(cu->curParser, TOKEN_ASSIGN)) {
489             COMPILE_ERROR(cu->curParser, "constructor shouldn't be setter!");
490         }
491
492         //构造函数必须是标准的 method,即 new(_,...),new 后面必须接(
493         if (!matchToken(cu->curParser, TOKEN_LEFT_PAREN)) {
494                 COMPILE_ERROR(cu->curParser, "constructor must be method!");
495         }
496
497         sign->type = SIGN_CONSTRUCT;
498
499         //无参数就直接返回
500         if (matchToken(cu->curParser, TOKEN_RIGHT_PAREN)) {
501             return;
502         }
503
504      } else {  //若不是构造函数
```

```
505
506             if (trySetter(cu, sign)) {
507                 //若是 setter,此时已经将 type 改为了 setter,直接返回
508                 return;
509             }
510
511             if (!matchToken(cu->curParser, TOKEN_LEFT_PAREN)) {
512                 //若后面没有(说明是 getter,type 已在开头置为 getter,直接返回
513                 return;
514             }
515
516                 //至此 type 应该为一般形式的 SIGN_METHOD,形式为 name(paralist)
517             sign->type = SIGN_METHOD;
518             //直接匹配到),说明形参为空
519             if (matchToken(cu->curParser, TOKEN_RIGHT_PAREN)) {
520                 return;
521             }
522         }
523
524         //下面处理形参
525         processParaList(cu, sign);
526         consumeCurToken(cu->curParser, TOKEN_RIGHT_PAREN, "expect ')' after
            parameter list!");
527 }
528
529 //为单运算符方法创建签名
530 static void unaryMethodSignature(CompileUnit* cu UNUSED, Signature* sign UNUSED) {
531     //名称部分在调用前已经完成,只修改类型
532     sign->type = SIGN_GETTER;
533 }
534
535 //为中缀运算符创建签名
536 static void infixMethodSignature(CompileUnit* cu, Signature* sign) {
537     //在类中的运算符都是方法,类型为 SIGN_METHOD
538     sign->type = SIGN_METHOD;
539
540     // 中缀运算符只有一个参数,故初始为 1
541     sign->argNum = 1;
542       consumeCurToken(cu->curParser, TOKEN_LEFT_PAREN, "expect '(' after infix operator!");
543     consumeCurToken(cu->curParser, TOKEN_ID, "expect variable name!");
544                 declareVariable(cu, cu->curParser->preToken.start, cu->
                    curParser->preToken.length);
545      consumeCurToken(cu->curParser, TOKEN_RIGHT_PAREN, "expect ')' after parameter!");
546 }
547
548 //为既做单运算符又做中缀运算符的符号方法创建签名
549 static void mixMethodSignature(CompileUnit* cu, Signature* sign) {
550     //假设是单运算符方法,因此默认为 getter
551     sign->type = SIGN_GETTER;
552
```

```
553        //若后面有(，说明其为中缀运算符，那就置其类型为 SIGN_METHOD
554        if (matchToken(cu->curParser, TOKEN_LEFT_PAREN)) {
555            sign->type = SIGN_METHOD;
556            sign->argNum = 1;
557                consumeCurToken(cu->curParser, TOKEN_ID, "expect variable name!");
558                        declareVariable(cu, cu->curParser->preToken.start, cu->
                        curParser->preToken.length);
559            consumeCurToken(cu->curParser, TOKEN_RIGHT_PAREN, "expect ')' after parameter!");
560        }
561 }
562
```

第 454 行定义的 trySetter 用于“尝试”编译 setter。setter 是包括=的方法，进入本函数时 preToken 是方法名。第 455 行判断若方法名后面没有=就返回 false，说明不是 setter。setter 有两种，第 459～463 行根据类型判断 setter 是 SIGN_SUBSCRIPT_SETTER 还是 SIGN_SETTER。第 466 行读取=右边形参列表的左小括号“(“。第 469～471 行读取形参并声明，注意此处不能调用 expression，原因是此处并不是赋值或函数调用等“实参”表达式，而是在编译方法声明，没有实际赋值的情况。第 474 行读取形参列表右边的小括号)。第 475 行使参数 argNum++，最后返回 true。

第 480 行定义的函数 idMethodSignature 是标识符的签名函数，为标识符创建签名。进入本函数时 preToken 是标识符，对于方法来说标识符即方法名，因此可能是 getter。第 482 行将签名 sign 设置为 SIGN_GETTER。第 485～504 行是处理构造函数的签名，构造函数是固定的方法名 new。构造函数后面肯定不能接=，因此第 488 行用来判断这一点。构造函数必须是 method，即形式如 new(_, _, ...)，因此第 493 行确保 new 后面必须是形参的左小括号“(”。随后第 497 行设置类型为构造函数 SIGN_CONSTRUCT。第 500 行判断构造函数的形参列表，若为空就直接返回。第 504～522 行用于处理非构造函数，此时第 506 行就要调用 trySetter，判断是不是 setter，若是则返回。此时 preToken 是方法名，第 511 行判断方法名后面是否跟有“(”，如果没有“(”说明是 getter，默认已经是 getter 了故直接返回。执行到第 516 行时，方法肯定为一般的 method，设置类型为 SIGN_METHOD。第 519 行判断形参列表，若为空就直接返回。最后第 525 行调用 processParaList 函数处理形参列表。第 526 行读取形参列表最右边的)。

第 530 行的函数 unaryMethodSignature 为前缀运算符方法创建签名，比如前缀运算符!作为数字类的方法时，表示对数字取反，sparrow 是完全面向对象的语言，数字运算要按照“对象.方法”来处理，如“!3”按照“对象.方法”调用会被编译为 3.!，这也正是“!”被视为类的方法的原因，而本函数就是为这种调用形式的前缀运算符生成签名。调用此函数前已经读取了方法名即前缀运算符，因此修改类型为 SIGN_GETTER 后直接返回。

第 536 行的函数 infixMethodSignature 为中缀运算符创建签名，同上，中缀运算符作为类的方法的原因也是要把中缀运算符（严谨来说是操作符，因此不限于数学）按照“对象.方法”的形式编译。如 1+2 会被按照“数字对象 1.+（数字对象 2）”的形式编译。本函数先是设置类型为 SIGN_METHOD，中缀运算符只有一个右操作数，后面读取只有一个操作数的形参列表并声明形参。

第 549 行的函数 mixMethodSignature 为既做前缀运算符又做中缀运算符的符号方法创建签名。用前缀运算符为方法名的方法形如 getter，因此第 551 行设置类型为 SIGN_GETTER。第 554 行判断若运算符后面有“(”，说明有参数列表，也就是中缀运算符的右操作数，因此那就置其类型为 SIGN_METHOD。中缀运算符只有一个右操作数，因此把 argNum 设置为 1。比如用+做方法，那么其在类中的定义形式为“+（形参）”。

6.11.3 upvalue 的查找与添加

upvalue 大家都清楚了，就是内层函数引用了外层函数中的局部变量，该局部变量对函数来说就是 upvalue。下面是它的查找与添加操作，代码如下。

stepByStep/c6/e/compiler/compiler.c(3/7)

```
563 //查找局部变量
564 static int findLocal(CompileUnit* cu, const char* name, uint32_t length) {
565     //内部作用域变量会覆外层,故从后往前,由最内层逐渐往外层找
566     int index = cu->localVarNum - 1;
567     while (index >= 0) {
568         if (cu->localVars[index].length == length &&
569             memcmp(cu->localVars[index].name, name, length) == 0) {
570             return index;
571         }
572         index--;
573     }
574     return -1;
575 }
576
577 //添加 upvalue 到 cu->upvalues,返回其索引，若已存在则只返回索引
578 static int addUpvalue(CompileUnit* cu, bool isEnclosingLocalVar, uint32_t index) {
579     uint32_t idx = 0;
580     while (idx < cu->fn->upvalueNum) {
581         //如果该 upvalue 已经添加过了就返回其索引
582         if (cu->upvalues[idx].index == index &&
583                                 cu->upvalues[idx].isEnclosingLocalVar ==
                                    isEnclosingLocalVar) {
584             return idx;
585         }
586         idx++;
587     }
588     //若没找到则将其添加
589         cu->upvalues[cu->fn->upvalueNum].isEnclosingLocalVar =
            isEnclosingLocalVar;
590     cu->upvalues[cu->fn->upvalueNum].index = index;
591     return cu->fn->upvalueNum++;
592 }
593
594 //查找 name 指代的 upvalue 后添加到 cu->upvalues,返回其索引,否则返回-1
595 static int findUpvalue(CompileUnit* cu, const char* name, uint32_t length) {
596     if (cu->enclosingUnit == NULL) { //如果已经到了最外层仍未找到,返回-1
597         return -1;
598     }
599     //进入了方法的 cu 并且查找的不是静态域,即不是方法的 Upvalue,那就没必要再往上找了
600     if (!strchr(name,' ') && cu->enclosingUnit->enclosingClassBK != NULL) {
601         return -1;
602     }
603
604     //查看 name 是否为直接外层的局部变量
```

```
605     int directOuterLocalIndex = findLocal(cu->enclosingUnit, name, length);
606
607     //若是,将该外层局部变量置为 upvalue,
608     if (directOuterLocalIndex != -1) {
609          cu->enclosingUnit->localVars[directOuterLocalIndex].isUpvalue = true;
610          return addUpvalue(cu, true, (uint32_t)directOuterLocalIndex);
611     }
612
613     //向外层递归查找
614     int directOuterUpvalueIndex = findUpvalue(cu->enclosingUnit, name, length);
615     if (directOuterUpvalueIndex != -1) {
616          return addUpvalue(cu, false, (uint32_t)directOuterUpvalueIndex);
617     }
618
619     //执行到此说明没有该 upvalue 对应的局部变量,返回-1
620     return -1;
621 }
622
623 //从局部变量和 upvalue 中查找符号 name
624 static Variable getVarFromLocalOrUpvalue(CompileUnit* cu, const char* name,
    uint32_t length) {
625     Variable var;
626     //默认为无效作用域类型,查找到后会被更正
627     var.scopeType = VAR_SCOPE_INVALID;
628
629     var.index = findLocal(cu, name, length);
630     if (var.index != -1) {
631          var.scopeType = VAR_SCOPE_LOCAL;
632          return var;
633     }
634
635     var.index = findUpvalue(cu,  name, length);
636     if (var.index != -1) {
637          var.scopeType = VAR_SCOPE_UPVALUE;
638     }
639     return var;
640 }
641
```

第 564 行定义的函数 findLocal 用于查找局部变量，我们在前面说过，localVars[localVarNum-1] 永远是当前作用域（也是目前最深层作用域）中最后一个局部变量，在内层作用域中定义的变量会屏蔽外层同名变量，因此从内层往外层逐层查找。找到后返回索引，找不到就返回-1。

我们对数据的管理很多都是相同的策略——用数组管理，用数组个数作为添加下一个数据的索引。比如局部变量放在数组 localVars 中，用 localVarNum 记录局部变量的个数并用其来添加局部变量。upvalue 也是这样的，用数组 upvalues 管理本编译单元中的 upvalue，用 upvalueNum 记录 upvalues 的个数并用其来添加 upvalue。

第 578 行定义的 addUpvalue 用于添加 upvalue 到 cu->upvalues，upvalue 在此为一对儿数据：“形参 isEnclosingLocalVar 表示 upvalue 是否为直接外层编译单元中的局部变量，如果是，那么形

参 index 表示此 upvalue 是直接外层编译单元中的局部变量的索引。如果不是，那么形参 index 表示此 upvalue 是直接外层编译单元中的 upvalue 的索引”。本函数先检查此 upvalue 是否已存在，如存在则返回其索引，否则添加后再返回索引。第 579～587 行从 cu 中遍历是否已存在相同索引的 upvalue，找到后第 584 行返回其索引。若未找到则将 upvalue 添加到 cu->upvalues 数组中，cu->fn->upvalueNum 用来记录 upvalue 的数量，同时它也是下一个 upvalue 的索引，第 589～590 行将其插入后，第 591 行更新 upvalueNum 为下一个待插入的位置并返回。

第 595 行定义的 findUpvalue 函数用于查找 name 指代的 upvalue 后添加到 cu->upvalues，返回其索引，否则返回-1。编译单元是可以嵌套的，通过成员 enclosingUnit 指向直接外层编译单元（父编译单元）实现嵌套链接形成链。位于编译单位中的局部变量可能被内层或更深内层引用为 upvalue，那么在查找 upvalue 时需要逐层查找，因此 findUpvalue 需要递归调用，在遍历过程中会把 upvalue 添加到编译单元链上所有的中间层。第 596～599 行判断若 enclosingUnit 为 NULL，说明到了最外层，递归结束。静态域是作为模块中的局部变量存在的，即我们把类中的静态域定义为模块中的局部变量。模块中所有类的静态域都会定义在同一模块中，因此为确保不同类中的静态域同名时不冲突，以“Cls 类名 静态域”的形式作为模块中的局部变量名。我们按照上面形式组合后，空格就成为中间的部分了，因此在内部，标识符中包括空格就是静态域的名字。另外前面已经说过了，如果 cu->enclosingUnit->enclosingClassBK !=NULL，说明 cu 是一个类中方法的编译单元。静态域是在模块编译单元中声明的局部变量。cu->enclosingUnit->enclosingClassBK != NULL 则表示 cu 是方法的编译单元。虽然在方法中声明的局部变量可被其内部定义的函数引用，但是进入到方法的 cu 时，此处查找的 name 并不是局部变量，name 对该方法来说也是 upvalue。理由很明显，你看，首先本函数就是用来查找 upvalue 的，其次，查找 name 时，都是在 cu 的直接外层即 cu->enclosingUnit 中查找，因此必然是视 name 为 upvalue。

方法的编译单元再上层就是模块的编译单元了（中间并没有类编译单元，前面说过，类没有编译单元），方法中的 upvalue 必然是模块中的局部变量。第 600 行判断，如果进入了方法的 cu 并且查找的不是静态域（即模块中的局部变量），那就没必要再往上遍历了，name 肯定是不存在的，因此可结束递归返回-1，即未找到。下面就是递归的过程。第 605 行调用 findLocal 判断形参 name 表示的 upvalue 是否为直接外层的局部变量，如果返回值 directOuterLocalIndex 不为-1，说明是直接外层的局部变量。第 609～610 行把直接外层中相应局部变量的 isUpvalue 置为 true，之前所说的“isUpvalue 是由内层编译单元（函数）设置”就是指此处，然后第 610 行调用 addUpvalue 把该 upvalue 添加到自己的编译单元中。如果该 upvalue 并不是直接外层函数的局部变量，有可能是直接外层函数的 upvalue，也有可能该 upvalue 对应的局部变量不存在，接下来沿 cu->enclosingUnit 向上递归调用 findUpvalue，如果最终在 x 层找到对应的局部变量，就将该 upvalue 添加到所有中间层编译单元的 upvalue 数组 upvalues 中。遍历结束后，结果是除（x-1）层 cu->upvalues[n].isEnclosingLocalVar 为 true 外，其他中间层 isEnclosingLocalVar 为 false。第 607 行递归调用 findUpvalue 在 cu 的直接外层查找，如果 directOuterUpvalueIndex 不为-1，说明找到了，那么就调用 addUpvalue 将其添加到本编译单元 cu 中，这就是在开头所说的，在所有中间层编译单元中都添加该 upvalue。如果该 upvalue 对应的局部变量存在，则一定会在第 589 行调用 findLocal 时找到该局部变量的索引，这就是正常的递归结束。如果执行到了第 613 行，说明没有该 upvalue 对应的局部变量，返回-1。

第 624 行的 getVarFromLocalOrUpvalue 用于从局部变量和 upvalue 中查找符号 name。实现也比较简单，先是调用 findLocal 从局部变量中查找，如果找到就返回变量。否则调用 findUpvalue 从 upvalue 中查找，如果找到就返回 var，否则就返回默认类型为 VAR_SCOPE_INVALID 的 var。这是非法类型，如果找到变量的话，此类型会被置为正确的类型，这在主调方中会判断。

6.11.4　变量的加载与存储

本小节完成变量读写，代码如下。

stepByStep/c6/e/compiler/compiler.c(4/7)

```
642 //生成把变量 var 加载到栈的指令
643 static void emitLoadVariable(CompileUnit* cu, Variable var) {
644     switch (var.scopeType) {
645         case VAR_SCOPE_LOCAL:
646             //生成加载局部变量入栈的指令
647                 writeOpCodeByteOperand(cu, OPCODE_LOAD_LOCAL_VAR, var.index);
648             break;
649         case VAR_SCOPE_UPVALUE:
650             //生成加载 upvalue 到栈的指令
651                 writeOpCodeByteOperand(cu, OPCODE_LOAD_UPVALUE, var.index);
652             break;
653         case VAR_SCOPE_MODULE:
654             //生成加载模块变量到栈的指令
655             writeOpCodeShortOperand(cu, OPCODE_LOAD_MODULE_VAR, var.index);
656             break;
657         default:
658             NOT_REACHED();
659     }
660 }
661
662 //为变量 var 生成存储的指令
663 static void emitStoreVariable(CompileUnit* cu, Variable var) {
664     switch (var.scopeType) {
665         case VAR_SCOPE_LOCAL:
666             //生成存储局部变量的指令
667                 writeOpCodeByteOperand(cu, OPCODE_STORE_LOCAL_VAR, var.index);
668             break;
669         case VAR_SCOPE_UPVALUE:
670             //生成存储 upvalue 的指令
671                 writeOpCodeByteOperand(cu, OPCODE_STORE_UPVALUE, var.index);
672             break;
673         case VAR_SCOPE_MODULE:
674             //生成存储模块变量的指令
675               writeOpCodeShortOperand(cu, OPCODE_STORE_MODULE_VAR, var.index);
676             break;
677         default:
678             NOT_REACHED();
679     }
680 }
681
682 //生成加载或存储变量的指令
683 static void emitLoadOrStoreVariable(CompileUnit* cu, bool canAssign, Variable var) {
684     if (canAssign && matchToken(cu->curParser, TOKEN_ASSIGN)) {
685         expression(cu, BP_LOWEST);    //计算=右边表达式的值
686         emitStoreVariable(cu, var);   //为 var 生成赋值指令
```

```
687      } else {
688          emitLoadVariable(cu, var);    //为 var 生成读取指令
689      }
690 }
691
```

第 643 行定义的函数 emitLoadVariable 用于生成把变量 var 加载到栈的指令，其实就是读变量。此函数根据变量作用域的类型分别生成不同的 load 指令。

第 663 行定义的函数 emitStoreVariable 用于为变量 var 生成存储的指令，也就是写变量，同样是根据变量作用域的类型分别生成不同的 store 指令。值得注意的是，各种 load 指令都需要从栈顶获得操作数，因此要事先在栈顶准备好数据。

第 683 行定义的函数 emitLoadOrStoreVariable 用于生成加载或存储变量的指令。第 684 行用于判断若标识符后面有=就表示为变量赋值，即存储变量，先调用 expression 函数解析等号右边的赋值表达式（也就是把所赋的值加载到栈顶供 store 指令使用），随后调用 emitStoreVariable 生成存储变量的指令。否则就是加载变量，即调用 emitLoadVariable 生成加载变量的指令。

6.11.5 编译代码块及结束编译单元

包括在大括号中的代码就是代码块，比如函数体、方法体、方法的块参数。一个代码块就被视为一个独立的指令流，也就是被视为一个编译单元，当此代码块编译完毕后，就需要结束这个编译单元并做一些收尾工作，就像往口袋中装满东西后要扎上口一样。代码如下。

stepByStep/c6/e/compiler/compiler.c(5/7)

```
692 //生成把实例对象 this 加载到栈的指令
693 static void emitLoadThis(CompileUnit* cu) {
694     Variable var = getVarFromLocalOrUpvalue(cu, "this", 4);
695     ASSERT(var.scopeType != VAR_SCOPE_INVALID, "get variable failed!");
696     emitLoadVariable(cu, var);
697 }
698
699 //编译代码块
700 static void compileBlock(CompileUnit* cu) {
701     //进入本函数前已经读入了{
702     while (!matchToken(cu->curParser, TOKEN_RIGHT_BRACE)) {
703        if (PEEK_TOKEN(cu->curParser) == TOKEN_EOF) {
704            COMPILE_ERROR(cu->curParser, "expect '}' at the end of block!");
705        }
706        compileProgram(cu);
707     }
708 }
709
710 //编译函数或方法体
711 static void compileBody(CompileUnit* cu, bool isConstruct) {
712     //进入本函数前已经读入了{
713     compileBlock(cu);
714     if (isConstruct) {
715         //若是构造函数就加载"this 对象"作为下面 OPCODE_RETURN 的返回值
716         writeOpCodeByteOperand(cu, OPCODE_LOAD_LOCAL_VAR, 0);
717     } else {
```

```
718             //否则加载 null 占位
719             writeOpCode(cu, OPCODE_PUSH_NULL);
720         }
721
722         //返回编译结果,若是构造函数就返回 this,否则返回 null
723         writeOpCode(cu, OPCODE_RETURN);
724 }
725
726 //结束 cu 的编译工作,在其外层编译单元中为其创建闭包
727 #if DEBUG
728 static ObjFn* endCompileUnit(CompileUnit* cu,
729                              const char* debugName, uint32_t debugNameLen) {
730     bindDebugFnName(cu->curParser->vm, cu->fn->debug, debugName, debugNameLen);
731 #else
732 static ObjFn* endCompileUnit(CompileUnit* cu) {
733 #endif
734     //标识单元编译结束
735     writeOpCode(cu, OPCODE_END);
736     if (cu->enclosingUnit != NULL) {
737         //把当前编译的 objFn 作为常量添加到父编译单元的常量表
738         uint32_t index = addConstant(cu->enclosingUnit,OBJ_TO_VALUE
                (cu->fn));
739
740         //内层函数以闭包形式在外层函数中存在,
741         //在外层函数的指令流中添加“为当前内层函数创建闭包的指令"
742         writeOpCodeShortOperand(cu->enclosingUnit,
                OPCODE_CREATE_CLOSURE, index);
743
744         //为 vm 在创建闭包时判断引用的是局部变量还是 upvalue,
745         //下面为每个 upvalue 生成参数
746         index = 0;
747         while (index < cu->fn->upvalueNum) {
748             writeByte(cu->enclosingUnit,
749                       cu->upvalues[index].isEnclosingLocalVar ? 1 : 0);
750             writeByte(cu->enclosingUnit,
751                       cu->upvalues[index].index);
752             index++;
753         }
754     }
755
756     ///下调本编译单元,使当前编译单元指向外层编译单元
757     cu->curParser->curCompileUnit = cu->enclosingUnit;
758     return cu->fn;
759 }
760
```

第 692 行定义的 emitLoadThis 函数用于生成把实例对象 this 加载到栈的指令。我们在 initCompileUnit 函数中已经把 this 存储到 localVars[0]，第 694 行调用函数 getVarFromLocalOrUpvalue 获得 this 到变量 var，然后第 696 行调用 emitLoadVariable 函数把 this 所在的 var 变量加载到栈顶。

第 700 行定义的 compileBlock 函数用于编译代码块。代码块是包括在一对大括号{代码}中的

代码。在本函数调用之前已经读取了左大括号“{”，本函数是处理大括号其余的部分，调用的就是 compileProgram 函数，当然本函数还是空的，将来完成它你会发现，这也会调用 compileBlock，总之，为了实现文法中无穷的句子，编译器离不开递归。编译完代码块后由 while 的循环条件读取右大括号“}”，到此代码块完成。

第 711 行定义的 compileBody 函数用于编译函数或方法体，形参 isConstruct 表示是否为构造函数的方法体。方法体就是代码块，只不过要额外多做一些工作。进入本函数前已经读入了“{”。第 697 行调用函数 compileBlock 编译代码块，下面判断若方法是构造函数，就要生成加载实例对象的指令，把实例对象作为后面操作码 OPCODE_RETURN 的返回值，即构造函数要返回一个新建的对象。这是通过第 716 行的操作码 OPCODE_LOAD_LOCAL_VAR 完成的，把“运行时栈”中的第 0 个局部变量加载到栈顶，我们规定，运行时栈中第 0 个局部变量用于存储实例对象，也就是说将来创建了实例对象也要按照此规定保存到此处。实例对象就是当前对象，如同 C++中的 this 指针，指向当前对象实例，其他语言中也有的叫 self。在我们这里也用 this 表示当前对象，不过我们语言中的 this 不是指针，我们是弱类型语言，无指针类型。因此第 716 行也可以用 emitLoadThis 函数代码，但 emitLoadThis 中要判断的东西太多，最终也是生成 OPCODE_LOAD_LOCAL_VAR 操作码，因此还不如此处更直接呢。这里有一个细节，此处是生成加载第 0 个局部变量（即 this 变量）到栈顶的指令，this 就是类的接收者（实例对象），它作为本函数最后 CODE_RETURN 的返回值。在调用本函数前肯定是预先生成了实例对象 this，并且把 this 对象放到了第 0 个局部变量中，要不是 OPCODE_LOAD_LOCAL_VAR，0 肯定加载不到 this 对象。注意，第 0 个局部变量并不是指 localVars[0]，此数组中的局部变量没有“值”，this 对象是个局部变量“值”，局部变量的值是存储到运行栈中，因此此处的第 0 个局部变量是指该方法运行时栈中的第 0 个局部变量，暂称为 stackStart[0]。提前说一下，负责生成 this 对象并把 this 对象提前放到 stackStart [0]中的函数就是 emitCreateInstance，希望将来遇到它时大家能想起今天的应用场景。第 717 行判断若不是构造函数，那么就往栈中压入 null 以保持栈平衡，也就是保证后面第 723 行的 OPCODE_RETURN 指令有参数可用，否则把栈顶的数据作为参数返回会导致栈失衡（栈失衡指的是栈中少了一个 slot 的数据，因此栈顶位置及栈大小就变了，后面依赖此栈中数据的指令必然会出错），因为操作码 OPCODE_RETURN 会把栈顶数据弹出，这里压入 NULL 当返回值就是返回空。

第 728 行定义的函数 endCompileUnit 用于结束 cu 的编译工作，在其外层编译单元中为其创建闭包。如果目前处于调试状态，会额外调用 bindDebugFnName，把函数名写到 fn->debug 中。结束一个编译单元的编译工作，首先要在该编译单元指令流中加上操作码 OPCODE_END 表示本编译单元的指令结束了，这就是第 735 行的作用。编译单元本质上是指令流单元，objFn 对象才是用于存储指令流的，因此编译单元被编译后的结果肯定是一个 objFn，如果编译单元 cu 有父编译单元，这说明 cu 是定义在别人的编译单元中的编译单元，这就是闭包，那么在第 738 行把编译单元的 objFn，即（cu->fn）作为常量添加到父编译单元的常量表，获得索引 index。第 742 行在父编译单元（直接外层函数）的指令流中添加“为常量 index 对应的内层函数创建闭包的指令”，也就是内层函数以闭包形式在外层函数中存在。说到闭包就离不开 upvalue，闭包就是内层函数，我们可以通过观察代码的方式分清内层函数引用的变量中，哪些是内层函数自己的局部变量，哪些是 upvalue。为什么要区分 upvalue 呢？因为 upvalue 是外层或更外层函数中的局部变量，真正的值是在外层函数的运行时栈中，此处的内层函数只是引用该值，在运行时为了获取 upvalue 的值，虚拟机需要到相应“层”中找到 upvalue 的“真身”，即对应的局部变量，然后才能拿到具体值，这个过程只能这样。但虚拟机在执行操作码 OPCODE_CREATE_CLOSURE 创建闭包时，它没有办法知道内层函数所引用的变量是否为 upvalue，只有在知道是 upvalue 后，才能去某外层访

问其对应的局部变量以获得真值。为解决这个问题，我们需要告诉虚拟机，在内层函数所引用的 upvalue 中，哪些是直接外层中的局部变量，哪些是直接外层中的 upvalue。下面第 744～754 行之间的 while 循环就是这个目的，为虚拟机在创建闭包时判断引用的是局部变量还是 upvalue，为每个 upvalue 生成一对儿数据，这对儿数据包括{upvalue 是否为直接外层的局部变量，upvalue 在直接外层中的索引}。每对儿数据都写入父编译单元（直接外层）的指令流中，虚拟机会从指令流中获取每对儿数据。第 757 行使 curCompileUnit = cu->enclosingUnit，即下调本编译单元 cu，使当前编译单元指向 cu 的外层编译单元，最后返回编译单元 cu 中的指令流 objFn，即 cu->fn。

我们将来在实现虚拟机时，最好是回来对照着看此处生成的指令，所有的指令都应该以“指令生成”和“虚拟机执行指令的细节”对照着看。“指令生成”和“虚拟机执行指令”的实现是相互配合的，它们互为合作方。比如编译器先生成指令并放在某处，这样虚拟机才知道去哪里获取指令。如果先介绍虚拟机，一定会有读者诧异：“为什么有的操作数要从栈中获取，有的操作数又在指令流中？指令是如何生成的？为什么要以这样的形式提供操作数？统一不好吗？”等一系列的问题，又会涉及本章所介绍的指令生成的内容，总之，无论先介绍哪个，都会因为另一个合作方还没介绍而产生迷惑。待虚拟机也完成后，你最好把各个指令的生成与执行对照着看一次，会茅塞顿开。

6.11.6　各种方法调用

stepByStep/c6/e/compiler/compiler.c(6/7)

```
761 //生成 getter 或一般 method 调用指令
762 static void emitGetterMethodCall(CompileUnit* cu, Signature*sign,OpCode opCode) {
763       Signature newSign;
764       newSign.type = SIGN_GETTER;   //默认为 getter,假设下面的两个 if 不执行
765       newSign.name = sign->name;
766       newSign.length = sign->length;
767       newSign.argNum = 0;
768
769      //如果是 method,有可能有参数列表，在生成调用方法的指令前必须把参数入栈，否则运行方法时除了
         //会获取到错误的参数(即栈中已有数据)外,g 还会在从方法返回时,错误地回收参数空间而导致栈失衡
770
771      //下面调用的 processArgList 是把实参入栈,供方法使用
772       if(matchToken(cu->curParser, TOKEN_LEFT_PAREN)) {        //判断后面是否有(
773          newSign.type = SIGN_METHOD;
774
775          //若后面不是),说明有参数列表
776          if (!matchToken(cu->curParser, TOKEN_RIGHT_PAREN)) {
777              processArgList(cu, &newSign);
778                  consumeCurToken(cu->curParser, TOKEN_RIGHT_PAREN, "expect ')'
                        after argument list!");
779          }
780       }
781
782       //对 method 来说可能还传入了块参数
783       if(matchToken(cu->curParser, TOKEN_LEFT_BRACE)) {
784          newSign.argNum++;
785          //进入本 if 块时,上面的 if 块未必执行过,
786            //此时 newSign.type 也许还是 GETTER,下面要将其设置为 METHOD
```

```
787         newSign.type = SIGN_METHOD;
788         CompileUnit fnCU;
789         initCompileUnit(cu->curParser, &fnCU, cu, false);
790
791         Signature tmpFnSign = {SIGN_METHOD, "", 0, 0}; //临时用于编译函数
792         if (matchToken(cu->curParser, TOKEN_BIT_OR)) {  //若块参数也有参数
793             processParaList(&fnCU, &tmpFnSign);      //将形参声明为函数的局部变量
794             consumeCurToken(cu->curParser,TOKEN_BIT_OR,"expect'|'after argument
                  list!");
795         }
796         fnCU.fn->argNum = tmpFnSign.argNum;
797
798         //编译函数体,将指令流写进该函数自己的指令单元 fnCu
799         compileBody(&fnCU, false);
800
801 #if DEBUG
802         //以此函数被传给的方法来命名这个函数, 函数名=方法名+" block arg"
803         char fnName[MAX_SIGN_LEN + 10] = {'\0'}; //"block arg\0"
804         uint32_t len = sign2String(&newSign, fnName);
805         memmove(fnName + len, " block arg", 10);
806         endCompileUnit(&fnCU, fnName, len + 10);
807 #else
808         endCompileUnit(&fnCU);
809 #endif
810     }
811
812     //如果是在构造函数中调用了 super 则会执行到此,构造函数中调用的方法只能是 super
813     if(sign->type == SIGN_CONSTRUCT) {
814         if (newSign.type != SIGN_METHOD) {
815             COMPILE_ERROR(cu->curParser, "the form of supercall is super()
                  or super(arguments)");
816         }
817         newSign.type = SIGN_CONSTRUCT;
818     }
819
820     //根据签名生成调用指令, 如果上面的三个 if 都未执行,此处就是 getter 调用
821     emitCallBySignature(cu, &newSign, opCode);
822 }
823
824 //生成方法调用指令,包括 getter 和 setter
825 static void emitMethodCall(CompileUnit* cu, const char* name,
826         uint32_t length, OpCode opCode, bool canAssign) {
827     Signature sign;
828     sign.type = SIGN_GETTER;
829     sign.name = name;
830     sign.length = length;
831
832     //若是 setter 则生成调用 setter 的指令
833     if(matchToken(cu->curParser, TOKEN_ASSIGN) && canAssign) {
834         sign.type = SIGN_SETTER;
```




```
835          sign.argNum = 1;    //setter 只接受一个参数
836
837          //载入实参(即=右边所赋的值),为下面方法调用传参
838          expression(cu, BP_LOWEST);
839
840          emitCallBySignature(cu, &sign, opCode);
841       } else {
842          emitGetterMethodCall(cu, &sign, opCode);
843       }
844 }
…略
```

第 726 行定义的函数 emitGetterMethodCall 用于生成 getter 或一般 method 调用指令，形参中的 sign 是待调用方法的签名，不过本函数是处理方法名后面的部分，也就是说默认本函数的主调方已经识别完了方法名，具体方法名的类型是 method 还是 getter 或 setter，是由本函数继续识别的，因此主调方提供的参数 sign 是不完整的，还需要本函数继续完善它，因此本函数并未充分使用 sign，而是在第 763～767 行生成签名结构 newSign，部分继承了形参 sign，这是为了避免破坏 sign->type，后面会根据 sign->type 是否为构造函数做相应的处理。第 772 行判断如果后面有左小括号“(”就说明为 method，在生成调用方法的指令前必须把参数入栈，否则运行方法时除了会获取到错误的参数（即栈中已有数据）外，还会从方法返回时错误地回收参数空间而导致栈失衡。第 776 行判断参数列表若不为空就在第 777 行调用函数 processArgList 处理实参。对 method 来说，参数还包括块参数。提醒，getter 是没有参数的，更谈不上块参数了。块参数就是传入的参数是一个代码块，即以函数为参数。注意，此处不是处理方法定义中的方法体，此处是处理方法调用的参数，不是方法定义！块参数就是夹在{代码块}中的代码块，因此块参数是以大括号“{”为标志。块参数是个函数，函数可以有参数，块参数的参数是夹在|形参列表|中的形参列表，没用过 ruby 的读者可能对块参数比较陌生，举个例子。

```
[work@work spr]$ cat -n /tmp/t.sp
    1class A {
    2    new() {
    3    }
    4    bar(fn) {
    5       fn.call(123)
    6    }
    7}
    8var a = A.new()
    9a.bar {|n| System.print(n)}
[work@work spr]$ ./spr /tmp/t.sp
123
[work@work spr]$
```

其中第 9 行调用 a.bar 时传入的{|n| System.print(n)}就是块参数。外形上看 a.bar 后面没有小括号而显得像是对 getter 的调用，但大家要清楚 getter 调用是不接参数的，强调一下，后接块参数的方法肯定是 SIGN_METHOD。第 783 行判断若包括块参数，就使参数 argNum++，因为我们把块参数当作 method 的实参来处理，因此块参数也是实参之一。第 787 行将类型设置为 SIGN_METHOD。第 788～789 行代码为编译块参数新建编译单元 fnCU 并将其初始化，注意，其父编译单元是 cu，这里就形成了编译单元的嵌套。临时新建的 tmpFnSign 用于记录块参数（函数）



中的参数个数。第 792～795 行声明块参数中的形参列表，函数 processParaList 会更新 tmpFnSign->argNum，处理完毕后第 796 行使块参数的参数个数为 tmpFnSign->argNum。第 799 行调用 compileBody 编译块参数的代码块，注意传入的编译单元是 fnCU，即把指令流写入到块参数自己的 objFn 中。编译结束后根据调试状态，调用合适的 endCompileUnit 以结束块参数编译单元的编译工作。本函数可以被多个函数调用，其中之一就是被关键字 super 的 nud 方法调用，该 nud 方法就是同名的 super（以后会实现此函数）。super 传给本函数的参数 sign 是关键字 super 所在的方法的签名，比如在方法 a 中调用了 super()，那么 sign 就是方法 a 的签名。在 sparrow 中 super 的作用是调用基类方法，也就是说 super 是用在子类方法中的关键字。super 有两种形式，一种是给出方法名，如 super.methodA()，这就是调用基类中的方法 methodA。如果未给出方法名，如 super()或 super（实参列表），就调用基类中的同名方法，如果是在子类的构造函数中调用 super，那么一定调用的是基类的构造函数，我们这里讨论的是后者未给出方法名的形式。但是要注意，super 的调用形式是 method，也就是说不能仅用关键字 super，而一定是 super()或 super（实参列表），因为构造函数不可能为 getter。时刻要清楚，本函数是假设已经由主调方读取了方法名，这里主调方是 super，那么已经由 super 函数读取了所调用方法 super()或 super（实参列表）中的 super 后才来调用本函数的，super 函数传给本函数的 sign 是关键字 super()或 super（实参列表）所在的方法的签名，并不是所调用方法 super()或 super（实参列表）的签名，因为我们要通过签名 sign 判断这两个方法的执行环境是否在构造函数中。第 813 行判断如果是在子类构造函数（方法名为 new）中调用了 super()或 super（实参列表），那么至少执行过上面任意某个 if 块才能使 newSign->type 为 SIGN_METHOD，这样才能确保 super 的调用形式不是 super 这种 getter 形式。随后第 817 行把 newSign->type 更新为 SIGN_CONSTRUCT，此时不用再更新 newSign->name，这里调用的是基类中同名方法。检查过后，此时被调用方法的签名 newSign 已经准备好，第 821 行调用函数 emitCallBySignature 为所调用的方法生成调用指令。

第 825 行的函数 emitMethodCall 用于生成方法调用指令，包括 getter 和 setter。这个函数就简单多了，先生成签名，将参数对号入座，第 833 行判断若是 setter 则生成调用 setter 的指令，否则就调用 emitGetterMethodCall 生成 getter 或 method 调用。

6.11.7 标识符的编译

我们建议：具有全局变量性质的变量名以大写字符开头，比如类名、模块变量名；具有局部变量性质的变量名以小写字符开头，比如方法名和局部变量名。由于以上只是对标识符命名规范的“建议”，用户不遵守也没关系，因此我们在语法分析过程中并未按照此规范去检查标识符的命名，脚本工作起来没出什么问题，不过有这么一种标识符冲突时判优的情况需要跟读者交代清楚。

我们知道，类中的方法有 3 种形式，分别是 getter、method 和 setter。这里所说的“冲突”是出现在 getter 和模块变量之间。getter 既然也是方法，那么调用 getter 方法的完整格式也是“对象名.方法名”，然而同一类中的方法之间是相互可见的，因此在方法中调用本类其他 getter 方法时就不需要再提供对象名了，只要提供方法名即可，编译系统会从当前对象所属类的方法中查找被调用的方法。this 就表示当前对象，因此，只提供方法名的方法调用相当于“this.方法调用”，我们也支持这样的代码，如果你愿意，可以在同类的方法调用前一律加上 this.（注意 this 后面有个“.”，this 就是当前对象，对象和方法之间一定有个“.”），我们的编译器可处理这种情况。注意，这里一直说的是 getter 方法，即调用时只需要提供方法名，不需要小括号及参数的那种方法。尽管这里讨论的是 getter 方法，但并不意味着这种调用形式只对 getter 方法适用，只要是调用同一类中的其他方法都不需要显式地加“对象.”。比如 setter、getter 和 method 方法都是对同类方法可见的，所以调用同类中任意一种方法都不需要显式地提供“对象.”，即若方法类型是 method，只



需要提供方法名及小括号围起的参数列表，如 xxx（参数列表）；若方法类型是 getter，只要提供方法名即可，如 xxx；若方法类型是 setter，只要提供方法名、=和实参即可，如“xxx=实参”。在方法中也可以引用模块变量，问题也就来了，由于模块变量不属于任何一个对象，因此引用模块变量时也只需要提供变量名即可，即引用形式与调用本类 getter 方法的形式是一样的，那么在类的方法中，若引用的模块变量名和同类中的其他 getter 方法名同名时，编译器认为引用的是模块变量还是 getter 方法调用？也就是说，编译器如何判断所引用的标识符是模块变量名还是本类的其他 getter 方法呢？这个必须要分清楚，因为编译器为这两类标识符生成的目标代码是不一样的，必须先把这两者区分开来，然后才能准确地为它们生成相应的目标代码。为解决这个冲突，我们规定：以小写字符开头的就是方法名。因此如果是模块变量，建议变量名以大写字母开头，如果模块变量以小写字符开头，当与类中的 getter 方法同名时，我们会优先选择 getter 方法。如果不冲突，无论模块变量名首字符是否大小写都可以引用到。补充一下，这里只是说 getter 方法名与模块变量名冲突时的择优，在类中优先级最高的是域，即类属性，本节后面介绍的 id 函数会处理标识符同名冲突时择优的优先级，当然这是后话。为说明眼前的问题，请看下面的样例代码。

```
[work@work spr]$ cat -n /tmp/1.sp
     1class foo {
     2    var name
     3    new(n) {
     4      name = n
     5    }
     6
     7    myName {
     8      return name
     9    }
    10
    11    hi() {
    12        System.print("hello,everyone, I am "
    13          + myName + ". My sister is " + MyName)
    14    }
    15}
    16
    17var f = foo.new("xiao ming")
    18var MyName = "da hong"
    19f.hi()
[work@work spr]$ ./spr /tmp/1.sp
hello,everyone, I am xiao ming. My sister is da hong
[work@work spr]$
[work@work spr]$ cat -n /tmp/2.sp
     1class foo {
     2    var name
     3    new(n) {
     4      name = n
     5    }
     6
     7    myName {
     8      return name
     9    }
    10
```

```
    11    hi() {
    12        System.print("hello,everyone, I am " + myName)
    13    }
    14}
    15
    16var f = foo.new("xiao ming")
    17var myName = "da hong"
    18f.hi()
[work@work spr]$ ./spr /tmp/2.sp
hello,everyone, I am xiao ming
[work@work spr]$
```

以上的两个示例代码中，类属于模块变量，具有全局性质，如果按照建议来命名，源码文件1.sp 中的类名应该是 Foo，这里故意以小写 foo 作为类名，是想让大家知道，以大写字符开头的命名规范确实只是建议，我们并没有强迫用户，因此不做检查。类中有一个名为 myName 的 getter 方法，此方法会返回域 name 的值。同时在类外也有一个名为 MyName 的模块变量，其值为 dahong。在 method 方法 hi 中，myName 的开头是小写字符，因此被处理为同类中的方法调用，MyName 是以大写字符开头，因此被处理为模块变量。大家看下运行结果，应该明白我的意思。示例文件2.sp 与 1.sp 有一点区别，就是模块变量名与 getter 方法同名，都是 myName，从程序运行的结果可以看出，编译器确实会优先选择 getter 方法。

下面介绍目前 compiler.c 的最后一个部分。标识符包括变量名、函数名等与定义相关的名称，标识符的 nud 方法用来编译该标识符，代码如下。

stepByStep/c6/e/compiler/compiler.c(7/7)

```
871 //小写字符开头便是局部变量
872 static bool isLocalName(const char* name) {
873     return (name[0] >= 'a' && name[0] <= 'z');
874 }
875
876 //标识符.nud():变量名或方法名
877 static void id(CompileUnit* cu, bool canAssign) {
878     //备份变量名
879     Token name = cu->parser->preToken;
880     ClassBookKeep* classBK = getEnclosingClassBK(cu);
881
882     //标识符可以是任意符号,按照此顺序处理:
883     //函数调用->局部变量和 upvalue->实例域->静态域->类 getter 方法调用->模块变量
884
885     //处理函数调用
886         if (cu->enclosingUnit == NULL && matchToken(cu->parser, TOKEN_LEFT_PAREN)) {
887         char id[MAX_ID_LEN] = {'\0'};
888
889         //函数名加上"Fn "前缀作为模块变量名,
890         //检查前面是否已有此函数的定义
891         memmove(id, "Fn ", 3);
892         memmove(id + 3, name.start, name.length);
893
894         Variable var;
895         var.scopeType = VAR_SCOPE_MODULE;
```

```
896         var.index = getIndexFromSymbolTable(
897             &cu->parser->curModule->moduleVarName, id, strlen(id));
898         if (var.index == -1) {
899             memmove(id, name.start, name.length);
900             id[name.length] = '\0';
901             COMPILE_ERROR(cu->parser, "Undefined function: '%s'!", id);
902         }
903
904 // 1. 把模块变量即函数闭包加载到栈
905         emitLoadVariable(cu, var);
906
907         Signature sign;
908         //函数调用的形式和 method 类似,
909         //只不过 method 有一个可选的块参数
910         sign.type = SIGN_METHOD;
911
912         //把函数调用编译为“闭包.call”的形式,故 name 为 call
913         sign.name = "call";
914         sign.length = 4;
915         sign.argNum = 0;
916
917         //若后面不是),说明有参数列表
918         if (!matchToken(cu->parser, TOKEN_RIGHT_PAREN)) {
919 // 2. 压入实参
920             processArgList(cu, &sign);
921             consumeCurToken(cu->parser, TOKEN_RIGHT_PAREN,
922                             "expect ')' after argument list!");
923         }
924
925 // 3. 生成调用指令以调用函数
926         emitCallBySignature(cu, &sign, OPCODE_CALL0);
927
928     } else {    //否则按照各种变量来处理
929
930         //按照局部变量和 upvalue 来处理
931         Variable var = getVarFromLocalOrUpvalue(cu,
932                        name.start, name.length);
933         if (var.index != -1) {
934             emitLoadOrStoreVariable(cu, canAssign, var);
935             return;
936         }
937
938         //按照实例域来处理
939         if (classBK != NULL) {
940             int fieldIndex = getIndexFromSymbolTable(&classBK->fields,
941                              name.start, name.length);
942             if (fieldIndex != -1) {
943                 bool isRead = true;
944                 if (canAssign && matchToken(cu->parser, TOKEN_ASSIGN)) {
945                     isRead = false;
```




```
946                     expression(cu, BP_LOWEST);
947                 }
948
949                 //如果当前正在编译类方法,则直接在该实例对象中加载 field
950                 if (cu->enclosingUnit != NULL) {
951                     writeOpCodeByteOperand(cu,
952                             isRead ?OPCODE_LOAD_THIS_FIELD:OPCODE_STORE-
                            THIS_FIELD, fieldIndex);
953                 } else {
954                     emitLoadThis(cu);
955                     writeOpCodeByteOperand(cu,
956                                 isRead ? OPCODE_LOAD_FIELD : OPCODE_
                                STORE_FIELD, fieldIndex);
957                 }
958                 return;
959             }
960         }
961
962         //按照静态域查找
963         if (classBK != NULL) {
964             char* staticFieldId = ALLOCATE_ARRAY(cu->parser->vm, char, MAX_ID_LEN);
965             memset(staticFieldId, 0, MAX_ID_LEN);
966             uint32_t staticFieldIdLen;
967             char* clsName = classBK->name->value.start;
968             uint32_t clsLen = classBK->name->value.length;
969
970             //各类中静态域的名称以 "Cls 类名 静态域名" 来命名
971             memmove(staticFieldId, "Cls", 3);
972             memmove(staticFieldId + 3, clsName, clsLen);
973             memmove(staticFieldId + 3 + clsLen, " ", 1);
974             const char* tkName = name.start;
975             uint32_t tkLen = name.length;
976             memmove(staticFieldId + 4 + clsLen, tkName, tkLen);
977             staticFieldIdLen = strlen(staticFieldId);
978                     var = getVarFromLocalOrUpvalue(cu, staticFieldId,
                        staticFieldIdLen);
979
980                     DEALLOCATE_ARRAY(cu->parser->vm, staticFieldId, MAX_ID_LEN);
981             if (var.index != -1) {
982                 emitLoadOrStoreVariable(cu, canAssign, var);
983                 return;
984             }
985         }
986
987         //如果以上未找到同名变量,有可能该标识符是同类中的其他方法调用
988         //方法规定以小写字符开头
989         if (classBK != NULL && isLocalName(name.start)) {
990             emitLoadThis(cu); //确保 args[0]是 this 对象,以便查找到方法
991             //因为类可能尚未编译完,未统计完所有方法,
992             //故此时无法判断方法是否为未定义,留待运行时检测
```




```
 993                 emitMethodCall(cu, name.start,
 994                         name.length, OPCODE_CALL0, canAssign);
 995                 return;
 996             }
 997
 998             //按照模块变量处理
 999             var.scopeType = VAR_SCOPE_MODULE;
1000             var.index = getIndexFromSymbolTable(
1001                         &cu->parser->curModule->moduleVarName, name.start, name.length);
1002             if (var.index == -1) {
1003                 //模块变量属于模块作用域,若当前引用处之前未定义该模块变量,
1004                 //说不定在后面有其定义,因此暂时先声明它,待模块统计完后再检查
1005
1006                 //用关键字 fun 定义的函数是以前缀 Fn 后接函数名作为模块变量
1007                 //下面加上 Fn 前缀按照函数名重新查找
1008                 char fnName[MAX_SIGN_LEN + 4] = {'\0'};
1009                 memmove(fnName, "Fn ", 3);
1010                 memmove(fnName + 3, name.start, name.length);
1011                 var.index = getIndexFromSymbolTable(
1012                             &cu->parser->curModule->moduleVarName, fnName, strlen(fnName));
1013
1014                 //若不是函数名,那可能是该模块变量定义在引用处的后面,
1015                 //先将行号作为该变量值去声明
1016                 if (var.index == -1) {
1017                             var.index = declareModuleVar(cu->parser->vm,
                                    cu->parser->curModule,
1018                                                          name.start, name.length,
                                                             NUM_TO_VALUE(cu->parser->curToken.
                                                             lineNo));
1019                 }
1020             }
1021             emitLoadOrStoreVariable(cu, canAssign, var);
1022         }
1023 }
…略
1045 SymbolBindRule Rules[] = {
1046         /* TOKEN_INVALID*/              UNUSED_RULE,
1047         /* TOKEN_NUM */                 PREFIX_SYMBOL(literal),
1048         /* TOKEN_STRING */              PREFIX_SYMBOL(literal),
1049         /* TOKEN_ID */                    {NULL, BP_NONE, id, NULL, idMethodSignature},
1050 };
…略
```

第 872 行定义的函数 isLocalName 用于判断是否为局部变量名，局部变量名以小写字符开头。

第 877 行定义的 id 函数是本节的主角，它是标识符的 nud 方法。进入 nud 方法时 preToken 是操作符即标识符，curToken 是操作符右边的符号。这里是在处理标识符的引用，并不是定义，也就是说标识符在之前已经定义好了，将来会看到各种标识符的定义过程。

第 879 行备份标识符，后面有用。当识别到一个标识符时，它可以是任意的符号，比如是函数名、变量名、域属性等，当变量名与域属性同名时，我们把该标识符当作什么呢？这就涉及优



先级顺序的问题，我们是按照此优先级处理的：函数调用->局部变量和 upvalue->实例域->静态域->类 getter 方法调用->模块变量。

通过 fun 关键字定义的函数只能在类外定义，如果在类内定义，就与类中的方法调用和方法定义混淆了，而且在方法中定义函数感觉很奇怪，毕竟可以用定义方法来代替。因此我们在语法上规定，对于用关键字 fun 定义的函数，只能在类外定义并且只在类外调用。如果非要在方法中定义函数，可以用 fn.new 的方式创建，然后用.call 的方式调用。第 886 行判断是否在类外，即 cu->enclosingUnit == NULL 表示类外，且是否是函数调用，即标识符后面的符号是“(”就是函数调用。如果是，就要检索该函数了。我们将来在处理用 fun 定义的函数时，会把函数名前加上前缀 Fn 后当作模块变量来存储，注意，Fn 后面是有空格的，这样是为了和用户定义的变量区分，因为用户代码中不可能会定义中间有空格的标识符，我们在词法分析时就会把它拆分了。第 887～902 行将函数名加上前缀 Fn 去检索，如果未找到就报错，如果找到了就在第 905 行将其加载到栈，后面马上要调用它。随后生成签名，函数调用是以“函数闭包.call”的形式，也就是说 call 是待调用的方法名，因此 sign.name 为 call。第 918～926 行处理实参列表并调用方法。如果标识符不是类外的函数调用，第 931 行先把标识符当作局部变量和 upvalue 来处理，如果从局部变量或 upvalue 中找到该标识符，那么调用 emitLoadOrStoreVariable 将其加载到栈后返回。

如果并不是局部变量或 upvalue，就把标识符按照实例域来处理。切记，此时并不是在处理标识符的定义，我们是在处理引用标识符，该标识符定义是在其他函数中完成，我们还没介绍呢。实例域是在类体中以“var 实例域名”来定义的变量。第 939 行判断若是在类中，就先从 classBK->fields 中获取域的信息，如果找到了该域，判断域后面是否有等号=，有就说明是在为域赋值。第 946 行调用 expression 解析赋值，结果留在栈顶，下面有用。如果是在类的方法中引用了域，也就是说引用了当前对象的域，那么就根据对域加载或存储的操作生成相应的指令。否则第 953 行按照在类外引用对象域来处理。上面第 946 行 expression 的结果用于操作码 OPCODE_STORE_THIS_FIELD 或 OPCODE_STORE_FIELD 的操作数。

如果并不是实例域就按照静态域处理，静态域是在类体中以“static var 静态域名”来定义的，第 964～977 行把静态域的名字按照“Cls 类名 静态域属性名”来组合，这样做的原因是，静态域是类属性，是被所有对象共享的数据，因此需要它长期有效，模块是最大的作用域，它是不会退出的，因此在模块的编译单元中定义类的静态域属性。同一个模块中可以有多个类，为了不同类中可以有同名的静态域属性，我们在定义静态域时是以“Cls 类名 静态域属性名”来命名的。因此此处的静态属性引用也要按照此格式来查找。

如果以上未找到，有可能该标识符是类中其他方法调用，比如 a 等同于调用了本类的 getter 方法 a，a()表示调用了本类的 method 方法 a()。举个例子。

```
[work@work spr]$ cat -n /tmp/t2.sp
     1class A {
     2    new() {
     3      pr()
     4    }
     5
     6    pr() {
     7        System.print("here")
     8    }
     9}
    10
    11var a=A.new()
```

```
[work@work spr]$ ./spr /tmp/t2.sp
here
[work@work spr]$
```

构造函数中的 pr()就是调用本类的方法 pr。

类名都是大写开头，因此若是小写字符开头便认为是类中的方法。换句话说，如果方法中使用了模块变量，则模块变量必须是大写字符开头才能使用，否则编译器会把标识符按照方法来处理，如果类中没有此方法，到运行时就会报错。第 989 行判断若是在类内且是方法，就调用函数 emitLoadThis 把对象 this 加载到栈，this 对象是给虚拟机准备的，一是用来填充 args[0]以确保 args[0]必须是 this 对象，二是用来确认对象所在的类，然后从类中获得所调用的方法。其实运行时栈 stackStart[0]是 this，但惯例上 args[0]应该是自身，即应该把 this 放到 args[0]，其他参数在数组 args 中依次排开，也就是说所调用方法的参数应该与 this 对象在栈中挨着才好，也就是说使 args[0]指向 stackStart[0]，然后 args[1]是方法左边第一个实参，其他以此类推。形参接近于栈底，与 stackStart[0]挨着，但实参就不能保证这一点，因为这是基于栈的虚拟机，栈顶是数据交换的中心，为方法传参时，实参也是放到栈顶，并不是放到形参所在的 slot 中，只有在方法中为形参赋值时才会往形参所在的 slot 中写数据。所以只好在需要 this 时把 this 加载到栈，后面再加载实参到栈顶以保持在栈中接壤，以使 args 数组满足惯例。后面第 993 行生成调用方法的指令，然后返回。

第 999 行，如果并不是同类中的方法调用，那么按照模块变量来查找，如果第 1002 行未找到该模块变量，即该模块变量未定义，则对于模块变量还有个宽大处理，模块变量属于模块中的全局变量，因此允许它定义在引用处的后面，如果在后面找到它的定义，那么语法上至少没问题，但引用处会用 null 值来代替。这里的策略是，暂时先用其所在的行号声明它，编译结束后再检查是否有其定义。不过在这之前还是要尝试把它作为函数名来查找看看，之前为标识符加上前缀 Fn 是在函数调用时的检查，现在是函数作为参数时的检查。比如函数名作为创建线程时的参数，如 thread.new（函数名），这种情况下要加上 Fn 前缀重新查找下。到了第 1016 行还是未找到，说明是未定义的模块变量，先用行号作为值将其声明为模块变量，此处就是占位，最后编译完毕时还要检查是否已有定义，若没有就正好用行号报错。之前在介绍 defineModuleVar 时说了很多，就是为了此处的伏笔。最后调用 emitLoadOrStoreVariable 处理标识符的读写情况。

第 1049 行在 Rules 数组中添加标识符的处理函数。

6.12 编译内嵌表达式（本节源码 stepByStep/c6/f）

本节编译内嵌表达式，代码如下。

stepByStep/c6/f/compiler/compiler.c

```
...略
871 //生成加载类的指令
872 static void emitLoadModuleVar(CompileUnit* cu, const char* name) {
873     int index = getIndexFromSymbolTable(
874                     &cu->curParser->curModule->moduleVarName, name, strlen(name));
875     ASSERT(index != -1, "symbol should have been defined");
876     writeOpCodeShortOperand(cu, OPCODE_LOAD_MODULE_VAR, index);
877 }
878
```



```
879 //内嵌表达式.nud()
880 static void stringInterpolation(CompileUnit* cu, bool canAssign UNUSED) {
881       // a %(b+c) d %(e) f
882       // 会按照以下形式编译
883       // ["a ", b+c, " d ", e, "f "].join()
884       // 其中 a 和 d 是 TOKEN_INTERPOLATION, b,c,e 都是 TOKEN_ID, f 是 TOKEN_STRING
885
886        //创造一个 list 实例,拆分字符串,将拆分出的各部分作为元素添加到 list
887        emitLoadModuleVar(cu, "List");
888        emitCall(cu, 0, "new()", 5);
889
890 //每次处理字符串中的一个内嵌表达式,包括两部分,以 a %(b+c) 为例:
891 //    1 加载 TOKEN_INTERPOLATION 对应的字符串,如 a,将其添加到 list
892 //    2 解析内嵌表达式,如 b+c,将其结果添加到 list
893     do {
894         // 1 处理 TOKEN_INTERPOLATION 中的字符串,如 a %(b+c) 中的 a
895         literal(cu, false);
896         //将字符串添加到 list
897         emitCall(cu, 1, "addCore_(_)", 11);   //以_结尾的方法名是内部使用
898
899         // 2 解析内嵌表达式,如 a %(b+c) 中的 b+c
900         expression(cu, BP_LOWEST);
901         //将结果添加到 list
902         emitCall(cu, 1, "addCore_(_)", 11);
903      } while (matchToken(cu->curParser, TOKEN_INTERPOLATION));
          // 处理下一个内嵌表达式,如 a %(b+c) d %(e) f 中的 d %(e)
904
905      //读取最后的字符串, a %(b+c) d %(e) f 中的 f
906      consumeCurToken(cu->curParser, TOKEN_STRING, "expect string at the end of
         interpolatation!");
907
908      //加载最后的字符串
909      literal(cu, false);
910
911      //将字符串添加到 list
912      emitCall(cu, 1, "addCore_(_)", 11);
913
914      //最后将以上 list 中的元素 join 为一个字符串
915      emitCall(cu, 0, "join()", 6);
916 }
…略
1092 SymbolBindRule Rules[] = {
1093      /* TOKEN_INVALID*/                    UNUSED_RULE,
1094      /* TOKEN_NUM */                       PREFIX_SYMBOL(literal),
1095      /* TOKEN_STRING */                    PREFIX_SYMBOL(literal),
1096      /* TOKEN_ID */                          {NULL, BP_NONE, id, NULL,
                                                  idMethodSignature},
1097                              /* TOKEN_INTERPOLATION */PREFIX_SYMBOL(stringInter
                                polation),
1098 };
```




第 872 行的函数 emitLoadModuleVar 用于生成加载模块变量的指令，类是以模块变量的形式存储的，基本上我们用此函数来加载类。

第 880 行的函数 stringInterpolation 用于编译内嵌表达式，注释已经足够清楚了，比较容易，不说了。

最后第 1097 行通过宏 PREFIX_SYMBOL 把 stringInterpolation 注册到 Rules 数组中。

6.13 编译 bool 及 null（本节源码 stepByStep/c6/g）

本节编译 true、false 及 null，代码如下。

stepByStep/c6/g/compiler/compiler.c

```
…略
945 //编译 bool
946 static void boolean(CompileUnit* cu, bool canAssign UNUSED) {
947       //true 和 false 的 nud 方法
948       OpCode opCode = cu->curParser->preToken.type ==
949          TOKEN_TRUE ? OPCODE_PUSH_TRUE : OPCODE_PUSH_FALSE;
950       writeOpCode(cu, opCode);
951 }
952
953 //生成 OPCODE_PUSH_NULL 指令
954 static void null(CompileUnit* cu, bool canAssign UNUSED) {
955       writeOpCode(cu, OPCODE_PUSH_NULL);
956 }
…略
1132 SymbolBindRule Rules[] = {
1133       /* TOKEN_INVALID*/                    UNUSED_RULE,
1134       /* TOKEN_NUM */                         PREFIX_SYMBOL(literal),
1135       /* TOKEN_STRING */                    PREFIX_SYMBOL(literal),
1136       /* TOKEN_ID */                          {NULL, BP_NONE, id, NULL,
                                                  idMethodSignature},
1137       /* TOKEN_INTERPOLATION */          PREFIX_SYMBOL(stringInterpolation),
1138       /* TOKEN_VAR */                         UNUSED_RULE,
1139       /* TOKEN_FUN */                         UNUSED_RULE,
1140       /* TOKEN_IF */                          UNUSED_RULE,
1141       /* TOKEN_ELSE */                        UNUSED_RULE,
1142       /* TOKEN_TRUE */                        PREFIX_SYMBOL(boolean),
1143       /* TOKEN_FALSE */                       PREFIX_SYMBOL(boolean),
1144       /* TOKEN_WHILE */                       UNUSED_RULE,
1145       /* TOKEN_FOR */                         UNUSED_RULE,
1146       /* TOKEN_BREAK */                       UNUSED_RULE,
1147       /* TOKEN_CONTINUE */                    UNUSED_RULE,
1148       /* TOKEN_RETURN */                      UNUSED_RULE,
1149       /* TOKEN_NULL */                        PREFIX_SYMBOL(null),
1150       /* TOKEN_CLASS */                       UNUSED_RULE,
1151 };
…略
```




6.14 this、继承、基类（本节源码 stepByStep/c6/h）

本节注册 this 对象，继承符号“<”和基类 super 的编译。

stepByStep/c6/h/compiler/compiler.c

```
...略
874 //"this".nud()
875 static void this(CompileUnit* cu, bool canAssign UNUSED) {
876     if (getEnclosingClassBK(cu) == NULL) {
877         COMPILE_ERROR(cu->curParser, "this must be inside a class method!");
878     }
879     emitLoadThis(cu);
880 }
881
882 //"super".nud()
883 static void super(CompileUnit* cu, bool canAssign) {
884     ClassBookKeep* enclosingClassBK = getEnclosingClassBK(cu);
885     if (enclosingClassBK == NULL) {
886         COMPILE_ERROR(cu->curParser, "can't invoke super outside a class method!");
887     }
888
889     //此处加载 this,是保证参数 args[0]始终是 this 对象,尽管对基类调用无用
890     emitLoadThis(cu);
891
892     //判断形式 super.methodname()
893     if (matchToken(cu->curParser, TOKEN_DOT)) {
894         consumeCurToken(cu->curParser, TOKEN_ID, "expect name after '.'!");
895         emitMethodCall(cu, cu->curParser->preToken.start,
896                        cu->curParser->preToken.length, OPCODE_SUPER0, canAssign);
897     } else {
898         //super():调用基类中与关键字 super 所在子类方法同名的方法
899         emitGetterMethodCall(cu, enclosingClassBK->signature, OPCODE_
                SUPER0);
900     }
901 }
...略
1162 SymbolBindRule Rules[] = {
1163     /* TOKEN_INVALID*/                  UNUSED_RULE,
1164     /* TOKEN_NUM */                       PREFIX_SYMBOL(literal),
1165     /* TOKEN_STRING */                  PREFIX_SYMBOL(literal),
1166     /* TOKEN_ID */                        {NULL, BP_NONE, id, NULL,
                                               idMethodSignature},
1167     /* TOKEN_INTERPOLATION */         PREFIX_SYMBOL(stringInterpolation),
1168     /* TOKEN_VAR */                       UNUSED_RULE,
1169     /* TOKEN_FUN */                       UNUSED_RULE,
1170     /* TOKEN_IF */                        UNUSED_RULE,
1171     /* TOKEN_ELSE */                      UNUSED_RULE,
1172     /* TOKEN_TRUE */                      PREFIX_SYMBOL(boolean),
1173     /* TOKEN_FALSE */                     PREFIX_SYMBOL(boolean),
```




```
1174        /* TOKEN_WHILE */                   UNUSED_RULE,
1175        /* TOKEN_FOR */                     UNUSED_RULE,
1176        /* TOKEN_BREAK */                   UNUSED_RULE,
1177        /* TOKEN_RETURN */                  UNUSED_RULE,
1178        /* TOKEN_NULL */                    PREFIX_SYMBOL(null),
1179        /* TOKEN_CLASS */                   UNUSED_RULE,
1180        /* TOKEN_THIS */                    PREFIX_SYMBOL(this),
1181        /* TOKEN_STATIC */                  UNUSED_RULE,
1182        /* TOKEN_IS */                      INFIX_OPERATOR("is", BP_IS),
1183        /* TOKEN_SUPER */                   PREFIX_SYMBOL(super),
1184 };
…略
```

第 875 行定义的 this 就是在程序中引用关键字 this 时编译器所调用的编译函数，脚本中的 this 就表示 this 对象，本函数调用 emitLoadThis 加载 this 对象到栈顶。

第 883 行定义的 super 就是在程序中引用关键字 super 时编译器所调用的编译函数，脚本中的 super 就表示基类，第 884～887 行用于检测使用关键字 super 的环境，不能在类外。第 890 行调用 emitLoadThis 加载 this，这个并不是为调用基类做准备，而是为在虚拟机中保证 args[0]是 this 对象，即调用 super 的那个子类对象，保证参数数组 args 一致。super 调用有两种方式，一种方式是指定基类中的方法，形式如 super.methodName(argList)，其中 methodName 是基类中的方法。另一种是调用与关键字 super 所在方法同名的基类方法，形式如 super(argList)。第 893～900 行便是处理这两种情况。

最后在第 1180～1183 行更新 Rules 数组，注册方法。

6.15 编译小括号、中括号及 list 列表字面量（本节源码 stepByStep/c6/i）

小括号对儿()用于提高优先级，用小括号括起来的表达式相当于被分为一组，作为一个整体参与其他运算。中括号对儿[]有两个用途。一是实现列表字面量形式的定义，字面量也称为直接量，是相对于变量来说，表面即是结果，不需要再去变量中取值，因此字面量比较直接，也称直接量。比如列表字面量的定义形式。

```
var listA= ["duang", 1+2*3, 'x']
//listA[0]为 duang
//listA[1]为 7
//listA[2]为 x
```

listA 就是拥有这 3 个元素的列表，此用途中的“[”不关注左操作数，因此是用'['.nud 方法来实现字面量列表定义。当然我们还可以采用添加的方式往列表中添加元素，代码如下。

```
[work@work spr]$ cat -n /tmp/t.sp
     1var a = List.new()
     2a.add(1+2*3)
     3System.print(a[0])
[work@work spr]$ ./spr /tmp/t.sp
7
[work@work spr]$
```

[]的另一用途是作为下标运算符，用来索引元素，比如从列表中索引元素，索引值为数字，如 listA[0]就是字符串 duang，此用途中的“[”关注左操作数 listA，因此是用“[”的 led 方法实现列表元素索引。下标运算符[]不仅用在列表中，还可以用在 map 中，只不过索引值就不是数字了，而是 key，到编译 map 时再说。本节编译小括号、中括号及列表字面量定义。

stepByStep/c6/i/compiler/compiler.c

```
...略
936 //编译圆括号
937 static void parentheses(CompileUnit* cu, bool canAssign UNUSED) {
938       //本函数是'('.nud(), 假设 curToken 是(
939       expression(cu, BP_LOWEST);
940       consumeCurToken(cu->curParser, TOKEN_RIGHT_PAREN,
941             "expect ')' after expression!");
942 }
943
944 //'['.nud() 处理用字面量形式定义的 list 列表
945 static void listLiteral(CompileUnit* cu, bool canAssign UNUSED) {
946 //进入本函数后,curToken 是[右面的符号
947
948       //先创建 list 对象
949       emitLoadModuleVar(cu, "List");
950       emitCall(cu, 0, "new()", 5);
951
952       do {
953          //支持字面量形式定义的空列表
954          if (PEEK_TOKEN(cu->curParser) == TOKEN_RIGHT_BRACKET) {
955             break;
956          }
957          expression(cu, BP_LOWEST);
958          emitCall(cu, 1, "addCore_(_)", 11);
959       } while (matchToken(cu->curParser, TOKEN_COMMA));
960
961       consumeCurToken(cu->curParser, TOKEN_RIGHT_BRACKET, "expect ')'
          after list element!");
962 }
963
964 //'['.led()  用于索引 list 元素,如 list[x]
965 static void subscript(CompileUnit* cu, bool canAssign) {
966
967       //确保[]之间不空
968       if (matchToken(cu->curParser, TOKEN_RIGHT_BRACKET)) {
969          COMPILE_ERROR(cu->curParser, "need argument in the '[]'!");
970       }
971
972       //默认是[_],即 subscript getter
973       Signature sign = {SIGN_SUBSCRIPT, "", 0, 0};
974
975       //读取参数并把参数加载到栈,统计参数个数
976       processArgList(cu, &sign);
977       consumeCurToken(cu->curParser,
```

```
978             TOKEN_RIGHT_BRACKET, "expect ']' after argument list!");
979
980       //若是[_]=(_),即 subscript setter
981       if (canAssign && matchToken(cu->curParser, TOKEN_ASSIGN)) {
982           sign.type = SIGN_SUBSCRIPT_SETTER;
983
984           //=右边的值也算一个参数,签名是[args[1]]=(args[2])
985           if (++sign.argNum > MAX_ARG_NUM) {
986               COMPILE_ERROR(cu->curParser,"the max number of argument is %d!",
                  MAX_ARG_NUM);
987           }
988
989           //获取=右边的表达式
990           expression(cu, BP_LOWEST);
991       }
992       emitCallBySignature(cu, &sign, OPCODE_CALL0);
993 }
994
995 //为下标操作符[编译签名
996 static void subscriptMethodSignature(CompileUnit* cu, Signature* sign) {
997       sign->type = SIGN_SUBSCRIPT;
998       sign->length = 0;
999       processParaList(cu, sign);
1000      consumeCurToken(cu->curParser,
1001            TOKEN_RIGHT_BRACKET, "expect ']' after index list!");
1002      trySetter(cu, sign);    //判断]后面是否接=为 setter
1003 }
…略
1231 SymbolBindRule Rules[] = {
…略
1252      /* TOKEN_SUPER */                   PREFIX_SYMBOL(super),
1253      /* TOKEN_IMPORT */                  UNUSED_RULE,
1254      /* TOKEN_COMMA */                    UNUSED_RULE,
1255      /* TOKEN_COMMA */                    UNUSED_RULE,
1256      /* TOKEN_LEFT_PAREN */              PREFIX_SYMBOL(parentheses),
1257      /* TOKEN_RIGHT_PAREN */             UNUSED_RULE,
1258      /* TOKEN_LEFT_BRACKET */             {NULL, BP_CALL, listLiteral,
          subscript), subscriptMethodSignature},
1259      /* TOKEN_RIGHT_BRACKET */           UNUSED_RULE,
1260 };
…略
```

第 937 行定义的 parentheses 用于编译()，进入本函数时 curToken 是运算符（右边的符号，即本函数假设已经读取了“(”。下面调用 expression 解析括号中间的表达式。最后第 940 行读入右小括号“)”。

第 945 行定义的 listLiteral 用于编译字面量形式定义的列表，即定义形式如[逗号分隔的元素列表]，本函数是左中括号“[”的 nud 方法，因此 preToken 是“[”，即已经读取了“[”。强调，此时是在处理定义列表，并不是索引列表元素。以字面量形式定义列表其实是一种语法糖，语法糖就是表面上提供一种更方便用户使用的形式，在内部的处理逻辑依然是不变的。就像小孩不喜欢吃饭时，妈妈把饭做成小动物形状孩子就喜欢吃了。

新建列表的核心方式是 List.new()，字面量形式新建（定义）列表只是语法糖，内部也绕不过此核心方式，然后调用列表的内部方法 addCore_（注意不是上面的 add 方法，以_结尾的方法都是内部使用的方法，目前在编译上也没阻止用户使用内部方法）往列表中插入元素，字面量形式定义列表就按照以上两步完成。List 类是我们定义在 core.script.inc 文件中的，我们将来会在编译此文件时把所有类包括 List 类注册到模块变量中，下面第 949 行调用 emitLoadModuleVar 加载 List 类，然后第 950 行生成调用 new 方法的指令，即完成调用 List.new()。接下来的 do-while 循环处理以逗号分隔的列表元素，每调用 expression 获取一个列表元素后，调用 emitCall 函数生成调用 List 类的方法 addCore 并将其添加到列表对象中，说明一下，以_结束的函数都是内部使用的。最后第 961 行读取右中括号“]”。

第 965 行定义的 subscript 函数是“[”的 led 方法，即把[]用于下标运算符，用于索引 list 元素，如 list[x]。第 968～970 行确认索引元素不为空，否则报语法错误。用于下标的[]有两种形式，第一种是读列表元素如 list[x]，这是 getter 形式的下标。第二种是写列表元素如 list[x]=y，这是 setter 形式的下标，比 getter 多了等号=。只有在读取右中括号“]”后才能确认是否跟有=，因此第 973 行的签名默认为 SIGN_SUBSCRIPT，即 subscript getter。第 976 行处理索引，此处虽然是调用 processArgList 处理参数列表，但实际上我们只支持一个索引值，这种 list[x,..]多个索引值是非法的，因此 processArgList 也只是处理了一个索引值。

注意，签名是方法的外形，是语法分析时编译器把用户输入的代码形式识别为签名，比如脚本代码 list[0] = abc，编译器把这种形式的代码对应到 list 类中的方法签名是 [_]=(_)，对于 list 对象来说，左边的_表示 args[1]，右边的_表示 args[2]，而 args[0]便是实例对象 this。整体上的调用形式相当于 args[0].[args[1]]=(args[2])。你看，尽管用户代码是 list[0] = "abc"，但用户不需要显式给被赋的值加入括号形如 list[0] = ("abc")，这正是编译器的功劳。这有些像语法糖了，而且用户也觉得这是理所应当的事。总之，尽管大多数情况下用户代码和签名是一致的，但并不是用户一定按照签名来调用，得看编译器是把用户代码映射到哪种签名上。

第 981 行处理 setter 形式的下标，即形如 list[args[1]]=args[2]，setter 中肯定要有等号“=”。第 982 行将签名类型设置为 SIGN_SUBSCRIPT_SETTER。方法中接受的参数个数也有限制，第 985 行判断参数个数是否合法。args[2]也是方法的参数之一，因此第 985 行对 sign.argNum 执行了++。之后第 990 行调用 expression 获取 args[2]，即所赋的值。第 992 行调用 emitCallBySignature 生成调用 [_]=(_)方法的指令。

第 996 行定义的函数 subscriptMethodSignature 用于为下标操作符“[”编译签名。[]签名要么为 getter 形式的 SIGN_SUBSCRIPT，即为 [_]，要么为 setter 形式的 SIGN_SUBSCRIPT_SETTER，即为 [_]=(_)。

最后把圆括号和中括号的编译方法录入到 Rules 数组。

6.16 编译方法调用和 map 字面量（本节源码 stepByStep/c6/j）

我们先说方法调用。对于一种面向对象语言来说，方法调用是“对象.方法”的形式，也就是说“.”是方法调用的符号，它的左操作数是对象，右操作数是方法名，因此“.”有 led 方法。

stepByStep/c6/j/compiler.c

```
...略
915 //'.'.led() 编译方法调用,所有调用的入口
```

```
916 static void callEntry(CompileUnit* cu, bool canAssign) {
917     //本函数是'.'.led(),curToken 是 TOKEN_ID
918     consumeCurToken(cu->curParser,
919           TOKEN_ID, "expect method name after '.'!");
920
921     //生成方法调用指令
922     emitMethodCall(cu, cu->curParser->preToken.start,
923           cu->curParser->preToken.length, OPCODE_CALL0, canAssign);
924 }
…略
1016 //map 对象字面量
1017 static void mapLiteral(CompileUnit* cu, bool canAssign UNUSED) {
1018     //本函数是'{'.nud(), curToken 是 key
1019
1020     //Map.new()新建 map,为存储字面量中的 key->value 做准备
1021
1022     //先加载类,用于调用方法时从该类的 methods 中定位方法
1023     emitLoadModuleVar(cu, "Map");
1024     //再加载调用的方法,该方法将在上面加载的类中获取
1025     emitCall(cu, 0, "new()", 5);
1026
1027     do {
1028        //可以创建空 map
1029        if (PEEK_TOKEN(cu->curParser) == TOKEN_RIGHT_BRACE) {
1030           break;
1031        }
1032
1033        //读取 key
1034        expression(cu, BP_UNARY);
1035
1036        //读入 key 后面的冒号
1037        consumeCurToken(cu->curParser,
1038              TOKEN_COLON, "expect ':' after key!");
1039
1040        //读取 value
1041        expression(cu, BP_LOWEST);
1042
1043        //将 entry 添加到 map 中
1044        emitCall(cu, 2, "addCore_(_,_)", 13);
1045
1046     } while (matchToken(cu->curParser, TOKEN_COMMA));
1047
1048     consumeCurToken(cu->curParser, TOKEN_RIGHT_BRACE, "map literal should end
        with \')\'!");
1049 }
…略
1277 SymbolBindRule Rules[] = {
…略
1306     /* TOKEN_LEFT_BRACE */              PREFIX_SYMBOL(mapLiteral),
1307     /* TOKEN_RIGHT_BRACE */             UNUSED_RULE,
```

```
1308      /* TOKEN_DOT */                    INFIX_SYMBOL(BP_CALL, callEntry),
1309 };
```

第 916 行的函数 callEntry 用于编译函数调用。尽管“.”关注左操作数，即对象，但在调用本函数之前，对象已经由 id 函数加载到栈中了，也就是 args[0]表示的 this，因此 curToken 是“.”后面的方法名。第 918 行调用 consumeCurToken 读取方法名，然后调用 emitMethodCall 生成调用“对象.方法”的指令，提示一下，emitMethodCall 中会调用函数 processArgList 将实参压栈以填充参数数组的其他元素 args[0～]，以后我们在实现虚拟机时大家会理解。

map 是我们的哈希数组，哈希数组实现有两种方式，一种是先创建 map 再添加元素的方式，另一种是字面量的形式，如下。

```
[work@work spr]$ cat -n /tmp/t.sp
     1//第一种
     2var m = Map.new()
     3m["a"] = 1+2*3
     4System.print(m["a"])
     5
     6//第二种
     7var ml = {
     8    "a":1,
     9    "b":"lol"
    10}
    11System.print(ml["b"])
    12System.print({"a":1,"b":"lol"}["b"])
[work@work spr]$ ./spr /tmp/t.sp
7
lol
lol
[work@work spr]$
```

第一种是用 new 方法创建 map 对象后再用 map[key]=value 为其添加元素，其实这种添加元素方式的方法签名是签名 [_]=(_)。第二种是用大括号对儿{}括起的 key:value 字面量形式定义了一个 map，key 和 value 之间是用“:”分隔，每对儿 keyValue 之间用逗号“,”分隔。与第一种不同的是定义的同时完成了元素初始化工作，这种用法很常见。尽管之前例子中都用了 var 把创建的对象声明为一个变量，但那只是为了使代码清楚一些，其实并不需要这样，第 12 行演示了直接创建对象后作为参数的用法。

第二种是本节要编译的内容，它也是语法糖，在内部是按照第一种方式处理的。第 1017 行定义的函数 mapLiteral 是“{”的 nud 方法，“{”是 map 字面量定义的开始。类 Map 是定义在 core.script.inc 中的，编译此文件时会把 Map 作为模块变量存储，因此第 1023 先把 Map 加载到栈，然后第 1025 行生成调用 new 方法的指令，以完成 Map.new()。下面就是逐个处理大括号中以逗号为分隔的 key:value 对儿，在内部也是调用了 addCore_方法，不过与 List 类中的 addCore_只是重名而已，而且形参个数不同，不是同一个函数。最后第 1048 行调用 consumeCurToken 读入定义结束的右大括号“}”。

最后把这些加入到 Rules 数组中。

也许有读者注意到了，新增的函数有很多都是添加到 compiler.c 文件的前面，这么做的主要原因是它们会被很多函数调用，为避免写一些函数声明，就放到前面了。

6.17 编译数学运算符（本节源码 stepByStep/c6/k）

数学运算符有很多，一些运算符的功能可由某些运算符组合完成，比如 a+=b 就相当于 a=a+b，因此 sparrow 语言也只支持部分运算符，已经够用了，这些运算符是：+、-、*、/、%、&、|、～、>>、<<、&&、||、!、==、!=、>、>=、<、<=、=、?:，额外再支持范围运算符..，注意，里面有两个点，它类似于 Python 中的分片运算符“:”，只不过“:”表示的范围不包括尾部元素。而..表示的范围包括尾部元素，区间闭合。比如 0..2 表示 0、1、2。而 0:2 表示 0、1。

对于大部分数学运算符来说，其编译方法我们都可以用已完成的宏来注册，一会我们就知道了。有一些涉及布尔值而执行不同分支的运算符，比如||、&&和?:运算符，它们根据不同的条件执行不同的分支，这就要费点力气了。不同的分支实际上就是不同的代码块，分支一般分为 true 分支和 false 分支，||、&&和?:这 3 个运算符实际上都是 if-then-else 结构。a||b 相当于 if(!a) then {b}，a && b 相当于 if(a) then {b}，c? a:b 相当于 if(c) then {a} else {b}。if-then-else 本质上就是跳到不同的代码块，因此其实现离不开跳转指令。

表面上当条件为 true 时就要去执行 true 分支，当条件为 false 时就去执行 false 分支。可是在内部，无论是高级代码还是低级指令，它们在物理上都是接壤的，即 false 分支对应的代码块紧跟在 true 分支对应的代码块之后，这就涉及“分支入口和分支出口”，比如假设条件 true，执行了 true 分支后，应该跳过后面的整个 false 分支对应的代码块，即跳到 false 分支的出口，否则 true 和 false 都执行，逻辑就乱了。同样，对于条件为 false 时，也要跳过整个 true 分支的代码块，进入到 false 分支的入口，以避免错误地执行 true 分支中的逻辑。正确的流程如图 6-2 所示。

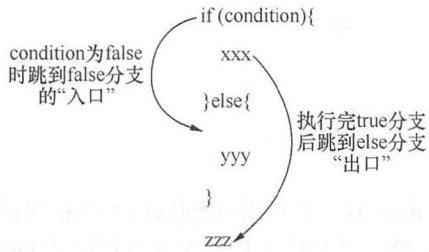


图 6-2

跳转是用类 jmp 指令来实现的，对于编译来说，除了把各分支中用户代码块原原本本地翻译为目标指令外，还要额外地插入不属于用户代码的跳转指令以跳过相应的分支，强调一下，这里说的是在各分支中插入跳转指令。

道理说得比较简单，可是各分支的出口和入口在哪里呢？也就是说我们跳转时的目标地址在哪里呢？目标地址取决于分支程序块的大小，跳转到目标时实际上就是跨过整个分支块的大小，在编译完 true 分支和 false 分支之前我们不知道这两个分支代码块的大小，因此在为各分支生成跳转指令时，指令的目标地址只能先空着，留待以后编译完各个分支后再回填。这就是编译原理中常说的回填技术，目标地址先空着，但基本上是用占位符来临时占位，这起到了标识的作用，以显示此处需要回填。

下面是代码。

stepByStep/c6/k/compiler/compiler.c （1/2）

```
...略
853 //用占位符作为参数设置指令
854 static uint32_t emitInstrWithPlaceholder(CompileUnit* cu, OpCode opCode) {
```

```
855     writeOpCode(cu, opCode);
856     writeByte(cu, 0xff);        //先写入高位的 0xff
857
858     //再写入低位的 0xff 后,减 1 返回高位地址,此地址将来用于回填
859     return writeByte(cu, 0xff) - 1;
860 }
861
862 //用跳转到当前字节码结束地址的偏移量去替换占位符参数 0xffff
863 //absIndex 是指令流中绝对索引
864 static void patchPlaceholder(CompileUnit* cu, uint32_t absIndex) {
865     //计算回填地址(索引)
866     uint32_t offset = cu->fn->instrStream.count - absIndex - 2;
867
868     //先回填地址高 8 位
869     cu->fn->instrStream.datas[absIndex] = (offset >> 8) & 0xff;
870
871     //再回填地址低 8 位
872     cu->fn->instrStream.datas[absIndex + 1] = offset & 0xff;
873 }
874
875 //'||'.led()
876 static void logicOr(CompileUnit* cu, bool canAssign UNUSED) {
877 //此时栈顶是条件表达式的结果,即||的左操作数
878
879     //操作码 OPCODE_OR 会到栈顶获取条件
880         uint32_t placeholderIndex = emitInstrWithPlaceholder(cu, OPCODE_OR);
881
882     //生成计算右操作数的指令
883     expression(cu, BP_LOGIC_OR);
884
885     //用右表达式的实际结束地址回填 OPCODE_OR 操作码的占位符
886     patchPlaceholder(cu, placeholderIndex);
887 }
888
889 //'&&'.led()
890 static void logicAnd(CompileUnit* cu, bool canAssign UNUSED) {
891 //此时栈顶是条件表达式的结果,即&&的左操作数
892
893     //操作码 OPCODE_AND 会到栈顶获取条件
894       uint32_t placeholderIndex = emitInstrWithPlaceholder(cu, OPCODE_AND);
895
896     //生成计算右操作数的指令
897     expression(cu, BP_LOGIC_AND);
898
899     //用右表达式的实际结束地址回填 OPCODE_AND 操作码的占位符
900     patchPlaceholder(cu, placeholderIndex);
901 }
902
```

第 854 行定义的 emitInstrWithPlaceholder 函数的作用是用占位符生成指令并返回占位符的地

址（指令流中的索引）。操作数是 2 字节，占位符通常是特征较明显的数字，我们这里是用 0x9999。大于 1 字节的数据就涉及字节序，我们一直都是大端字节序，因此在第 855 行写入操作码后，按照大端字节序先写入占位符的高字节，最后第 859 行写入低字节，写入低字节后 writeByte 返回的是操作数低字节在指令流中的地址，大端字节序高字节在前，因此减 1 便是操作数高字节地址，返回操作数高字节地址。

第 864 行定义的 patchPlaceholder 函数用于为占位符打补丁，也就是回填占位符的值，具体来说是用跳转到当前字节码结束地址的偏移量去替换占位符参数 0xffff，顺便说一下，偏移量都是正数，故往回跳转时是用“ip-=偏移量”实现的。注意，参数 absIndex 是指令流中的绝对地址，并不是相对的偏移量。第 866 行开始计算回填地址，我们现在所计算的偏移量 offset 是跳转指令的操作数，也就是将来虚拟机在执行到跳转的操作码时，会从指令流中获取到的操作数。虚拟机是靠指令指针 ip 获取指令中的操作码和操作数，按理说既然跳转指令的操作数是偏移量，那么直接用当前指令地址 cu->fn->instrStream.count 减去 absIndex 就行了，为什么还要减 2 呢？原因是，我们现在是在编译阶段，但是考虑的是运行时的情况，-2 是操作数本身占 2 字节，在执行跳转时已经读取了指令流中这 2 字节的偏移量，即 ip 已经+2 了，故 ip 与实际的目标地址便少了 2，因此此时把跳转偏移量减去 2 以修正。这种情况如图 6-3 所示。

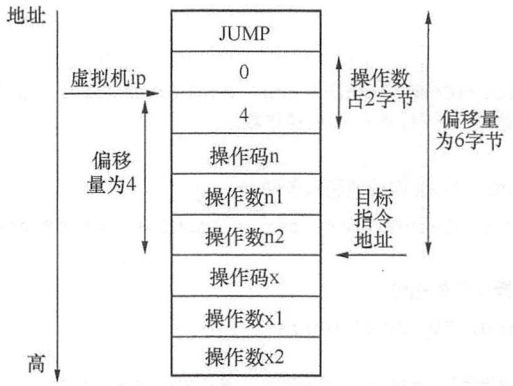


图 6-3

计算出了偏移量（即跳转指令的操作数）后，第 869 行先回填偏移量的高字节，第 872 行回填低字节。

第 876 行定义的函数 logicOr 是“逻辑或”运算符||的 led 方法，负责编译||表达式。||运算符是短路运算符，当左操作数为真时，右操作数就不用判断真假了。进入本函数时栈顶是条件表达式的结果，条件表达式即是||的左操作数。若条件（左操作数）为真则需要跳过||的右操作数的计算步骤，即跨过计算右操作数的指令步骤才行。不过，因为只有运行时才知道条件是否为真，而且尚不知道跳转的偏移量（即计算右表达式的指令流大小），此时先为偏移量设置占位符，编译好右操作数后再回填。第 880 行先用占位符作为操作码 OPCODE_OR 的操作数，返回值 placeholderIndex 是跳转指令的操作数高字节的索引，一会用来回填。||的右操作数必然也是个表达式，表达式包括数字、变量、函数调用等。之前说过，尽管 expression 是用来计算表达式，但其计算过程其实就是生成计算表达式步骤指令的过程，这就是编译的工作。第 883 行调用 expression 生成计算右操作数的一系列指令。在执行过 expression 后，我们已经知道了右操作数对应指令流的结束地址，即 cu->fn->instrStream.count 就是最后一条指令在指令流中的索引。此时我们可以回填若左操作数为真时需要跳过的偏移量了，因此第 886 行调用 patchPlaceholder 回填此偏移量。注意，尽管此时已经生成了计算右操作数的一系列指令，但有可能是用不上的。因为如果左操作数为真，那么用

于计算右操作数的那些指令就要被跳过，这是由操作码 OPCODE_OR 跳过的，将来看虚拟机的实现就明白了，正如本章开头所说，很多都基于将来的实现才能理解现在的内容。所以，并不是在编译时决定用哪个操作数，我们必须都要提前准备好两个操作数的计算指令步骤，在运行时决定。这在其他语言中也是一样的，无论执行哪个分支，分支中的代码必须要提前编译好，留待实际运行时选择。也许有读者觉得这样白白浪费了一个分支的编译，肯定有更好的办法在编译时就知道编译哪个分支以提高效率，也许有，不过我估计是没有，因为如果编译阶段就能知道走哪个分支，那程序还要运行干吗。

第 890 行定义的函数 logicAnd 是“逻辑与”运算符的 led 方法。&&运算符也是短路运算符，当左操作数为假时，右操作数就不用判断真假了，直接跳过。进入本函数时，栈顶是条件表达式的结果，即&&的左操作数，如果左操作数为假，就跳过&&右操作数的计算，同样只在运行时才知道条件是否为真，而且尚不知道跳转的偏移量（即计算右表达式的指令流大小），此时先为偏移量设置占位符，编译好右操作数后再回填。本函数逻辑和上面的 logicOr 相通，不再赘述，代码如下。

stepByStep/c6/k/compiler/compiler.c (2/2)

```
903 //"? :".led()
904 static void condition(CompileUnit* cu, bool canAssign UNUSED) {
905     //若 condition 为 false, if 跳转到 false 分支的起始地址,为该地址设置占位符
906         uint32_t falseBranchStart = emitInstrWithPlaceholder(cu,
            OPCODE_JUMP_IF_FALSE);
907
908     //编译 true 分支
909     expression(cu, BP_LOWEST);
910
911     consumeCurToken(cu->curParser, TOKEN_COLON, "expect ':' after true branch!");
912
913     //执行完 true 分支后需要跳过 false 分支
914         uint32_t falseBranchEnd = emitInstrWithPlaceholder(cu, OPCODE_JUMP);
915
916     //编译 true 分支已经结束.此时知道了 true 分支的结束地址,
917     //编译 false 分支之前须先回填 falseBranchStart
918     patchPlaceholder(cu, falseBranchStart);
919
920     //编译 false 分支
921     expression(cu, BP_LOWEST);
922
923     //知道了 false 分支的结束地址,回填 falseBranchEnd
924     patchPlaceholder(cu, falseBranchEnd);
925 }
…略
1352 SymbolBindRule Rules[] = {
…略
1384     /* TOKEN_DOT_DOT */                INFIX_OPERATOR("..", BP_RANGE),
1385     /* TOKEN_ADD */                      INFIX_OPERATOR("+", BP_TERM),
1386     /* TOKEN_SUB */                      MIX_OPERATOR("-"),
1387     /* TOKEN_MUL */                      INFIX_OPERATOR("*", BP_FACTOR),
1388     /* TOKEN_DIV */                      INFIX_OPERATOR("/", BP_FACTOR),
1389     /* TOKEN_MOD */                      INFIX_OPERATOR("%", BP_FACTOR),
```

仅供非商业用途或交流学习使用


```
1390        /* TOKEN_ASSIGN */                  UNUSED_RULE,
1391        /* TOKEN_BIT_AND */                 INFIX_OPERATOR("&", BP_BIT_AND),
1392        /* TOKEN_BIT_OR */                  INFIX_OPERATOR("|", BP_BIT_OR),
1393        /* TOKEN_BIT_NOT */                 PREFIX_OPERATOR("~"),
1394        /* TOKEN_BIT_SHIFT_RIGHT */         INFIX_OPERATOR(">>", BP_BIT_SHIFT),
1395        /* TOKEN_BIT_SHIFT_LEFT */          INFIX_OPERATOR("<<", BP_BIT_SHIFT),
1396        /* TOKEN_LOGIC_AND */               INFIX_SYMBOL(BP_LOGIC_AND, logicAnd),
1397        /* TOKEN_LOGIC_OR */                INFIX_SYMBOL(BP_LOGIC_OR, logicOr),
1398        /* TOKEN_LOGIC_NOT */               PREFIX_OPERATOR("!"),
1399        /* TOKEN_EQUAL */                   INFIX_OPERATOR("==", BP_EQUAL),
1400        /* TOKEN_NOT_EQUAL */               INFIX_OPERATOR("!=", BP_EQUAL),
1401        /* TOKEN_GREATE */                  INFIX_OPERATOR(">", BP_CMP),
1402        /* TOKEN_GREATE_EQUAL */            INFIX_OPERATOR(">=", BP_CMP),
1403        /* TOKEN_LESS */                    INFIX_OPERATOR("<", BP_CMP),
1404        /* TOKEN_LESS_EQUAL */              INFIX_OPERATOR("<=", BP_CMP),
1405        /* TOKEN_QUESTION */                INFIX_SYMBOL(BP_ASSIGN, condition),
1406        /* TOKEN_EOF */                       UNUSED_RULE
1407 };
…略
```

第 904 行定义的 condition 函数是条件运算符?:的 led 方法。条件表达式的形式如 condition ? true branch : false branch，相当于 if (condition) then {true branch} else {false branch}。进入本函数时，栈顶是 condition 的值。如果条件 condition 为假，就去执行 false 分支。此时尚不知道 true 分支代码块的大小，因此不知道 false 分支的入口地址，跳转指令的操作数就不能确定了，于是在第 906 行先调用函数 emitInstrWithPlaceholder 设置操作码 OPCODE_JUMP_IF_FALSE 的占位符，返回值 falseBranchStart 是 false 分支的入口地址。为什么此时先设置占位符而不是后面？原因很简单，跳转指令的操作数是偏移量，无论跳转目标是哪里，肯定都是当前编译单元指令流中的索引，即 cu->fn->instrStream.datas[index]是目标指令地址。编译是按照代码顺序从上到下，指令流的增长同样也是按照代码的顺序，判断条件 condition 真假值的跳转指令 OPCODE_JUMP_IF_FALSE 肯定位于 true 分支的指令流之前，这样才能保证跳过整个 true 分支进入 false 分支的指令流中，因此我们是在编译 true 分支前设置占位。第 909 行调用 expression 编译 true 分支，编译之后便知道了 true 分支的结束地址，此时应该在 true 分支结束处添加跳过 false 分支的跳转指令 OPCODE_JUMP，由于此时尚不知道 false 分支的结束地址，因此第 914 行先为操作码 OPCODE_JUMP 设置占位符。这是无条件跳转指令，即与 condition 无关，直接跳过整个 false 分支。注意，此时的 OPCODE_JUMP 指令是添加到了 true 分支中。编译完 true 分支后，也就是说知道了 true 分支的结束地址，那么也就知道了与其接壤的 false 分支的起始地址，于是第 918 行就可以回填前面 false 分支的起始地址了。下面第 921 行调用 expression 编译 false 分支，编译完成后知道了 false 分支的结束地址。第 924 行回填 falseBranchEnd。

第 1384～1406 行在 Rules 数组中新加了很多数学运算符，以及上面介绍过的逻辑与、逻辑或及条件运算符。

6.18 编译变量定义（本节源码 stepByStep/c6/1）

之前都是编译前的准备，本节开始正式进入编译环节，先从简单的开始，编译一个变量定义，即“var 变量名”，注意我们并不支持一次定义多个变量，即这种形式是错误的：“var 变量 1，变

仅供非商业用途或交流学习使用

量 2,...”。

stepByStep/c6/l/compiler/compiler.c

```
1462 //编译变量定义
1463 static void compileVarDefinition(CompileUnit* cu, bool isStatic) {
1464         consumeCurToken(cu->curParser, TOKEN_ID, "missing variable name!");
1465     Token name = cu->curParser->preToken;
1466     //只支持定义单个变量
1467     if (cu->curParser->curToken.type == TOKEN_COMMA) {
1468         COMPILE_ERROR(cu->curParser, "'var' only support declaring a variable.");
1469     }
1470
1471     //一、先判断是否是类中的域定义，确保 cu 是模块 cu
1472     if (cu->enclosingUnit == NULL && cu->enclosingClassBK != NULL) {
1473         if (isStatic) {   //静态域
1474             char* staticFieldId = ALLOCATE_ARRAY(cu->curParser->vm, char,
                    MAX_ID_LEN);
1475             memset(staticFieldId, 0, MAX_ID_LEN);
1476             uint32_t staticFieldIdLen;
1477             char* clsName = cu->enclosingClassBK->name->value.start;
1478             uint32_t clsLen = cu->enclosingClassBK->name->value.length;
1479
1480             //用前缀 "'Cls '+类名+变量名" 作为静态域在模块编译单元中的局部变量
1481             memmove(staticFieldId, "Cls", 3);
1482             memmove(staticFieldId + 3, clsName, clsLen);
1483             memmove(staticFieldId + 3 + clsLen, " ", 1);
1484             const char* tkName = name.start;
1485             uint32_t tkLen = name.length;
1486             memmove(staticFieldId + 4 + clsLen, tkName, tkLen);
1487             staticFieldIdLen = strlen(staticFieldId);
1488
1489             if (findLocal(cu, staticFieldId, staticFieldIdLen) == -1) {
1490                 int index = declareLocalVar(cu, staticFieldId,
                        staticFieldIdLen);
1491                 writeOpCode(cu, OPCODE_PUSH_NULL);
1492                 ASSERT(cu->scopeDepth == 0, "should in class scope!");
1493                 defineVariable(cu, index);
1494
1495                 //静态域可初始化
1496                 Variable var = findVariable(cu, staticFieldId,
                        staticFieldIdLen);
1497                 if (matchToken(cu->curParser, TOKEN_ASSIGN)) {
1498                     expression(cu, BP_LOWEST);
1499                     emitStoreVariable(cu, var);
1500                 }
1501
1502             } else {
1503                 COMPILE_ERROR(cu->curParser,
1504                     "static field '%s' redefinition!", strchr(staticFieldId,
                        ' ') + 1);
```

仅供非商业用途或交流学习使用


```
1505            }
1506
1507        } else {  //定义实例域
1508            ClassBookKeep* classBK = getEnclosingClassBK(cu);
1509
1510            int fieldIndex = getIndexFromSymbolTable(
1511                    &classBK->fields, name.start, name.length);
1512            if (fieldIndex == -1) {
1513                fieldIndex = addSymbol(cu->curParser->vm,
1514                        &classBK->fields, name.start, name.length);
1515
1516            } else {
1517                if (fieldIndex > MAX_FIELD_NUM) {
1518                    COMPILE_ERROR(cu->curParser,
1519                            "the max number of instance field is %d!",
                                MAX_FIELD_NUM);
1520                } else {
1521                    char id[MAX_ID_LEN] = {'\0'};
1522                    memcpy(id, name.start, name.length);
1523                    COMPILE_ERROR(cu->curParser,
1524                            "instance field '%s' redefinition!", id);
1525                }
1526
1527                if (matchToken(cu->curParser, TOKEN_ASSIGN)) {
1528                    COMPILE_ERROR(cu->curParser,
1529                            "instance field isn't allowed initialization!");
1530                }
1531            }
1532        }
1533        return;
1534    }
1535
1536    //二、若不是类中的域定义,就按照一般的变量定义
1537    if (matchToken(cu->curParser, TOKEN_ASSIGN)) {
1538        //若在定义时赋值就解析表达式,结果会留到栈顶
1539        expression(cu, BP_LOWEST);
1540    } else {
1541        //否则就初始化为 NULL,即在栈顶压入 NULL,
1542        //也是为了与上面显式初始化保持相同栈结构
1543        writeOpCode(cu, OPCODE_PUSH_NULL);
1544    }
1545
1546    uint32_t index = declareVariable(cu, name.start, name.length);
1547    defineVariable(cu, index);
1548 }
…略
```

第 1463 行定义的 compileVarDefinition 函数用于编译变量定义，参数 isStatic 表示是否在编译静态属性。调用本函数时已经读入了关键字 var，其实也正是因为主调方函数识别到了关键字 var 后才决定调用的本函数（主调方函数是 compileProgram），此时 curToken 应该是 var 后面的变量名。

仅供非商业用途或交流学习使用

通常在其他语言中一次定义多个变量时，变量间是以逗号分隔的，我们并不支持多个变量声明，必须是一个 var 声明一个变量。尽管不排除用户可能用其他符号作为多个变量间的分隔符，但其他字符会被词法分析器视为非法，因此第 1467 行简单判断标识符（变量名），后面若是逗号则报错。

我们在类中定义的属性（域）也是通过 var 关键字实现的，因此本函数除了处理一般的变量定义，也要处理属性定义的情况。第 1472 行先判断所编译的 var xxx 是否是类中的属性定义。cu->enclosingUnit == NULL 表示处于模块的编译单元，即编译器正在编译模块，后面在编译类定义时你会看到，如果正在编译方法体，即 cu 若是方法编译单元的话，那么 cu->enclosingUnit 的值应该为模块的编译单元，肯定不会为 NULL，此处先这么理解就行。当编译器编译一个类时，就会把 cu->enclosingClassBK 置为所编译的类的 classBookKeep 结构，故 cu->enclosingClassBK != NULL 表示正在编译类，这两个条件的综合就表示刚进入类，尚未进入类方法的编译单元，域定义正好是此阶段，这种情况如图 6-4 中的方框所示。

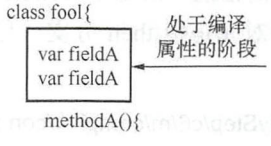


图 6-4

本函数的调用者会判断编译的是否是静态域，如果是静态域，形参 isStatic 的值就为 true。静态域和实例域有所不同，实例域只属于对象实例，只由实例自身访问，因此在存储上实例域只与所在对象相关。而静态属性被所有对象共享，因此在存储上要把它放在一个可被所有对象访问的地方，我们知道，局部变量出了作用域后就结束了生存周期，其空间会被回收，那么该局部变量就不存在了。模块是最大的作用域，它是不会退出的，因此我们把静态域作为模块的局部变量。不过对用户来说，在模块中定义的变量称为模块变量，用户是不能在模块中定义局部变量的。可是有一个问题，同一个作用域中只允许一个变量名存在，否则就重复定义了，那么在模块中定义多个类时，如果各类中的静态域同名，也会造成同名冲突，为解决此问题，对于静态域的存储，我们在内部用前缀“'Cls '+类名+静态域名”这种组合为变量名，以此变量名在模块编译单元中定义该局部变量，这就保证了多个类中的静态域不冲突。第 1473 行判断若是编译静态属性，就按照上面的策略把它定义为模块中的局部变量。第 1474～1478 行是为下面组合后的变量名申请缓冲区空间 staticFieldId。第 1481～1487 行把组合后的变量名存储到缓冲区空间 staticFieldId，变量名长度为 staticFieldIdLen。第 1489 行调用 findLocal 从模块编译单元中寻找以静态域对应的局部变量，如果找到，说明当前的静态域是重定义，那就在第 1503 行报错。如果找不到，说明是新定义的，那就把静态域在模块中声明并定义为局部变量。静态域属于静态数据，因此允许定义时赋值，第 1497 行处理这种情况。

如果形参 isStatic 为 false，说明是实例域，第 1507 行处理这种情况。实例域是存储在 ClassBookKeep->fields 中，首先在第 1508 行获取 cu 的 ClassBookKeep 结构 classBK，以后在编译类时你会知道，我们在编译静态方法时 classBK->inStatic 才会为 true。第 1510 行从 ClassBookKeep->fields 中获取实例域，如果找不到，就调用 addSymbol 将其插入。如果找到，那就是重定义，不过同时判断是否超过了最大的实例域个数。实例域仅属于对象的私有数据，定义时不可被初始化，因此第 1527 行判断后面若有等号就报错。

仅供非商业用途或交流学习使用

到了第 1537 行就是处理一般的变量定义了，一般变量定义是只可以初始化的，如果在标识符后面有=，那么就调用 expression 解析所赋的值，切记，我们是基于栈的虚拟机，那么虚拟机在执行 expression 所生成的用于计算表达式的一系列指令后，栈顶便是表达式值。如果定义时并没有初始化，那么在栈中压入 null，一是表示未初始化时变量为 Null，另一目的是为与上面显式初始化保持相同的栈结构，即栈顶得有个值。最后声明并定义这个变量。至于是局部变量还是模块变量，函数 declareVariable 和 defineVariable 中会判断。

6.19 编译语句

在 BNF 文法中，语句常用 STATEMENT 表示。

6.19.1 编译 if 语句（本节源码 stepByStep/c6/m）

本节编译 if 语句，if 语句在我们的语法中形如 if (condition) {} else {}。

注意，条件后面的第一个大括号对儿是指 then 分支（尽管没有关键字 then），then 分支就是 true 分支，else 分支就是 false 分支。

stepByStep/c6/m/compiler/compiler.c

```
...略
1551 //编译 if 语句
1552 static void compileIfStatement(CompileUnit* cu) {
1553     consumeCurToken(cu->curParser, TOKEN_LEFT_PAREN, "missing '(' after if!");
1554     expression(cu, BP_LOWEST);    //生成计算 if 条件表达式的指令步骤
1555     consumeCurToken(cu->curParser,
1556         TOKEN_RIGHT_PAREN, "missing ')' before '{' in if!");
1557
1558     //若条件为假，if 跳转到 false 分支的起始地址,现为该地址设置占位符
1559     uint32_t falseBranchStart =
1560        emitInstrWithPlaceholder(cu, OPCODE_JUMP_IF_FALSE);
1561
1562     //编译 then 分支
1563     //代码块前后的{和}由 compileStatement 负责读取
1564     compileStatement(cu);
1565
1566     //如果有 else 分支
1567     if(matchToken(cu->curParser, TOKEN_ELSE)) {
1568        //添加跳过 else 分支的跳转指令
1569          uint32_t falseBranchEnd = emitInstrWithPlaceholder(cu, OPCODE_JUMP);
1570
1571        //进入 else 分支编译之前,先回填 falseBranchStart
1572        patchPlaceholder(cu, falseBranchStart);
1573
1574        //编译 else 分支
1575        compileStatment(cu);
1576
1577        //此时知道了 false 块的结束地址,回填 falseBranchEnd
1578        patchPlaceholder(cu, falseBranchEnd);
1579
1580     }else {    //若不包括 else 块
```

仅供非商业用途或交流学习使用


```
1581          //此时 falseBranchStart 就是条件为假时,需要跳过整个 true 分支的目标地址
1582          patchPlaceholder(cu, falseBranchStart);
1583      }
1584 }
1585
1586 //编译语句(即程序中与声明,定义无关的、表示“动作”的代码)
1587 static void compileStatement(CompileUnit* cu) {
1588      if(matchToken(cu->curParser, TOKEN_IF)) {
1589          compileIfStatement(cu);
1590      }
1591 }
...略
```

第 1552 行定义的 compileIfStatement 函数用于编译 if 语句。进入本函数时，关键字 if 已经由主调方函数读取了，其实也正是因为主调方函数识别到了关键字 if 后才决定调用的本函数（主调方函数是 compileStatement），因此 curToken 是 if 后面的“(”。第 1553～1555 行读取 if 后面的(condition)。如果条件为假，那么虚拟机要跳过 then（即 true，下同）分支，直接进入 else（即 false，下同）分支，此时尚不知道 then 分支的结束地址，因此第 1559 行先占位，待编译完 then 分支后再回填。第 1564 行调用 compileStatement 函数编译 then 分支，compileStatement 函数是在下面实现，一会儿会说到。分支就是个代码块，代码块是由大括号{}包起来的代码集合。大括号对儿{}是由函数 compileStatement 直接间接读取的，故这里不用读取大括号。经过 compileStatement 后，then 分支代码块被编译完成，此时知道了 then 分支的结束地址。如果后面有 else 分支的话，接下来的便是 else 分支指令流的首地址，此时便可以回填 if 条件为假的跳转目标地址 falseBranchStart。第 1567 行处理 else 分支。如果 if 条件为真就要在执行了 then 分支后跳过 else 分支，为保证这一点，这需要在 then 分支的结尾添加无条件跳转指令 OPCODE_JUMP 以跳过整个 else 分支代码块，目前尚不知道 else 块的结束地址，因此在第 1569 行将 OPCODE_JUMP 的目标地址占位。在未编译 else 分支之前，第 1572 行回填 falseBranchStart。第 1575 行调用 compileStatement 编译 else 分支。编译 else 分支之后知道了 false 块的结束地址，回填当 if 为真时执行完 true 块后跳过 false 块的目标地址 falseBranchEnd。否则若没有 else 分支的话，此时已编译完 true 分支，知道了 true 分支的结束地址，因此在第 1582 行回填 if 条件为假时跳过 true 分支的目标地址 falseBranchStart。

第 1587 行定义的 compileStatement 函数用于编译语句，程序中的代码大致分为定义和语句两类，定义是指声明数据的代码，不具备能动性，相对来说“静态”一些，比如定义变量、定义类的代码。语句是指具备能动性的代码，可执行各种“动作”，比如 Symtem.print、return 及 break 等。compileStatement 便用于处理这类“动作”代码。目前本函数尚不完整，若识别到了 if 关键字，那么就调用上面介绍的函数 compileIfStatement 处理，此时已经读取了 if，因此进入 compileIfStatement 时的 curToken 便是 if 之后的符号了。

语句包含多种，不仅 if 是语句，while 也是语句，for 也是，代码块也是语句的集合，语句又可以嵌套，因此难免要递归调用 compileStatement。

6.19.2 编译 while 语句（本节源码 stepByStep/c6/n）

while 语句的语法如 while (condition) {循环体}，下面是其编译方法。

stepByStep/c6/n/compiler/compiler.c （1/2）

```
...略
1586 //开始循环,进入循环体的相关设置等
```

仅供非商业用途或交流学习使用


```
1587 static void enterLoopSetting(CompileUnit* cu, Loop* loop) {
1588     //cu->fn->instrStream.count 是下一条指令的地址,所以-1
1589     loop->condStartIndex = cu->fn->instrStream.count - 1;
1590
1591     loop->scopeDepth = cu->scopeDepth;
1592
1593     //在当前循环层中嵌套新的循环层,当前层成为内嵌层的外层
1594     loop->enclosingLoop = cu->curLoop;
1595
1596     //使 cu->curLoop 指向新的内层
1597     cu->curLoop = loop;
1598 }
1599
1600 //编译循环体
1601 static void compileLoopBody(CompileUnit* cu) {
1602     //使循环体起始地址指向下一条指令地址
1603     cu->curLoop->bodyStartIndex = cu->fn->instrStream.count;
1604
1605     compileStatment(cu);  //编译循环体
1606 }
1607
1608 //获得 ip 所指向的操作码的操作数占用的字节数
1609 uint32_t getBytesOfOperands(Byte* instrStream,
1610                             Value* constants, int ip) {
1611
1612     switch ((OpCode)instrStream[ip]) {
1613         case OPCODE_CONSTRUCT:
1614         case OPCODE_RETURN:
1615         case OPCODE_END:
1616         case OPCODE_CLOSE_UPVALUE:
1617         case OPCODE_PUSH_NULL:
1618         case OPCODE_PUSH_FALSE:
1619         case OPCODE_PUSH_TRUE:
1620         case OPCODE_POP:
1621             return 0;
1622
1623         case OPCODE_CREATE_CLASS:
1624         case OPCODE_LOAD_THIS_FIELD:
1625         case OPCODE_STORE_THIS_FIELD:
1626         case OPCODE_LOAD_FIELD:
1627         case OPCODE_STORE_FIELD:
1628         case OPCODE_LOAD_LOCAL_VAR:
1629         case OPCODE_STORE_LOCAL_VAR:
1630         case OPCODE_LOAD_UPVALUE:
1631         case OPCODE_STORE_UPVALUE:
1632             return 1;
1633
1634         case OPCODE_CALL0:
1635         case OPCODE_CALL1:
1636         case OPCODE_CALL2:
```

仅供非商业用途或交流学习使用


```
1637        case OPCODE_CALL3:
1638        case OPCODE_CALL4:
1639        case OPCODE_CALL5:
1640        case OPCODE_CALL6:
1641        case OPCODE_CALL7:
1642        case OPCODE_CALL8:
1643        case OPCODE_CALL9:
1644        case OPCODE_CALL10:
1645        case OPCODE_CALL11:
1646        case OPCODE_CALL12:
1647        case OPCODE_CALL13:
1648        case OPCODE_CALL14:
1649        case OPCODE_CALL15:
1650        case OPCODE_CALL16:
1651        case OPCODE_LOAD_CONSTANT:
1652        case OPCODE_LOAD_MODULE_VAR:
1653        case OPCODE_STORE_MODULE_VAR:
1654        case OPCODE_LOOP:
1655        case OPCODE_JUMP:
1656        case OPCODE_JUMP_IF_FALSE:
1657        case OPCODE_AND:
1658        case OPCODE_OR:
1659        case OPCODE_INSTANCE_METHOD:
1660        case OPCODE_STATIC_METHOD:
1661        return 2;
1662
1663        case OPCODE_SUPER0:
1664        case OPCODE_SUPER1:
1665        case OPCODE_SUPER2:
1666        case OPCODE_SUPER3:
1667        case OPCODE_SUPER4:
1668        case OPCODE_SUPER5:
1669        case OPCODE_SUPER6:
1670        case OPCODE_SUPER7:
1671        case OPCODE_SUPER8:
1672        case OPCODE_SUPER9:
1673        case OPCODE_SUPER10:
1674        case OPCODE_SUPER11:
1675        case OPCODE_SUPER12:
1676        case OPCODE_SUPER13:
1677        case OPCODE_SUPER14:
1678        case OPCODE_SUPER15:
1679        case OPCODE_SUPER16:
1680        //OPCODE_SUPERx 的操作数是分别由 writeOpCodeShortOperand
1681        //和 writeShortOperand 写入的,共 1 个操作码和 4 个字节的操作数
1682           return 4;
1683
1684        case OPCODE_CREATE_CLOSURE: {
1685           //获得操作码 OPCODE_CLOSURE 操作数, 2 字节
1686           //该操作数是待创建闭包的函数在常量表中的索引
```

仅供非商业用途或交流学习使用


```
1687            uint32_t fnIdx = (instrStream[ip + 1] << 8) | instrStream[ip + 2];
1688
1689            //左边第一个 2 是指 fnIdx 在指令流中占用的空间
1690            //每个 upvalue 有一对儿参数
1691            return 2 + (VALUE_TO_OBJFN(constants[fnIdx]))->upvalueNum * 2;
1692        }
1693
1694        default:
1695        NOT_REACHED();
1696    }
1697 }
```

第 1587 行的函数 enterLoopSetting 用于编译循环体之前的设置，这里用到了之前介绍的 Loop 结构。编译循环体之前要记录循环体的起始地址，这样循环体才能回到正确的起始位置实现往复执行，任何时候执行 enterLoopSetting，都是以当前指令为循环的开始地址，cu->fn->instrStream.count 是下一条指令的地址，减 1 是当前指令的地址。因此在第 1589 行把当前指令地址作为循环起始地址存储到 loop->condStartIndex。第 1591 行设置了循环体的作用域。第 1594～1597 行设置了循环体嵌套链，把循环 loop 的直接外层循环 loop->enclosingLoop 设置为当前循环层 cu->curLoop，然后使当前循环层 cu->curLoop 指向新的循环 loop。

第 1601 行的函数 compileLoopBody 用于编译循环体，第 1603 行先以下一条指令地址作为循环体的起始，然后第 1605 行调用 compileStatement 编译循环体。

第 1608 行定义的 getBytesOfOperands 函数用于获得 ip 所指向的操作码的操作数占用的字节数，也就是获得操作码对应操作数的字节大小。前面就不说了，介绍下操作码 OPCODE_CREATE_CLOSURE，其操作数是待创建闭包的函数在常量表中的索引，占 2 字节。此操作码是由函数 endCompileUnit 为编译单元 cu 创建闭包而生成的，闭包中可能涉及 upvalue，endCompileUnit 分别为函数中每个 upvalue 创建了一对儿参数，即 2 个参数，分别是布尔值“是否为直接外层的局部变量”和索引值“upvalue 在直接外层中的索引”。upvalue 的数量是存储在函数对象 objFn->upvalueNum。因此第 1691 行调用 VALUE_TO_OBJFN 先获取到该函数，再从中获取 upvalue 总数 upvalueNum，然后乘 2，这里左边加的那个 2 字节是指 fnIdx 在指令流中占用的空间。

stepByStep/c6/n/compiler/compiler.c (2/2)

```
1698
1699 //离开循环体时的相关设置
1700 static void leaveLoopPatch(CompileUnit* cu) {
1701     //获取往回跳转的偏移量,偏移量都为正数
1702     int loopBackOffset = cu->fn->instrStream.count - cu->curLoop->condStartIndex + 2;
1703
1704     //生成向回跳转的 CODE_LOOP 指令,即使 ip -= loopBackOffset
1705     writeOpCodeShortOperand(cu, OPCODE_LOOP, loopBackOffset);
1706
1707     //回填循环体的结束地址
1708     patchPlaceholder(cu, cu->curLoop->exitIndex);
1709
1710     //下面在循环体中回填 break 的占位符 OPCODE_END
1711     //循环体开始地址
1712     uint32_t idx = cu->curLoop->bodyStartIndex;
1713     //循环体结束地址
```

```
1714      uint32_t loopEndindex = cu->fn->instrStream.count;
1715      while (idx < loopEndindex) {
1716         //回填循环体内所有可能的 break 语句
1717         if (OPCODE_END == cu->fn->instrStream.datas[idx]) {
1718              cu->fn->instrStream.datas[idx] = OPCODE_JUMP;
1719              //回填 OPCODE_JUMP 的操作数,即跳转偏移量
1720
1721              //id+1 是操作数的高字节,patchPlaceholder 中会处理 idx + 1 和 idx + 2
1722              patchPlaceholder(cu, idx + 1);
1723
1724              //使 idx 指向指令流中下一操作码
1725              idx += 3;
1726         } else {
1727              //为提高遍历速度,遇到不是 OPCODE_END 的指令,
1728              //一次跳过该指令及其操作数
1729              idx += 1 + getBytesOfOperands(cu->fn->instrStream.datas,
1730                   cu->fn->constants.datas, idx);
1731         }
1732      }
1733
1734      //退出当前循环体, 即恢复 cu->curLoop 为当前循环层的外层循环
1735      cu->curLoop = cu->curLoop->enclosingLoop;
1736 }
1737
1738 //编译 while 循环,如 while (a < b) {循环体}
1739 static void compileWhileStatment(CompileUnit* cu) {
1740      Loop loop;
1741
1742     //设置循环体起始地址等等
1743      enterLoopSetting(cu, &loop);
1744      consumeCurToken(cu->curParser,
1745           TOKEN_LEFT_PAREN, "expect '(' befor condition!");
1746      //生成计算条件表达式的指令步骤,结果在栈顶
1747      expression(cu, BP_LOWEST);
1748      consumeCurToken(cu->curParser,
1749           TOKEN_RIGHT_PAREN, "expect ')' after condition!");
1750
1751      //先把条件失败时跳转的目标地址占位
1752           loop.exitIndex = emitInstrWithPlaceholder(cu, OPCODE_JUMP_IF_
                FALSE);
1753      compileLoopBody(cu);
1754
1755      //设置循环体结束等等
1756      leaveLoopPatch(cu);
1757 }
1758
1759 //编译语句(即程序中与声明、定义无关的,表示“动作”的代码)
1760 static void compileStatment(CompileUnit* cu) {
1761      if(matchToken(cu->curParser, TOKEN_IF)) {
1762         compileIfStatment(cu);
```

```
1763        }else if (matchToken(cu->curParser, TOKEN_WHILE)) {
1764            compileWhileStatment(cu);
1765        }
1766    }
```

第 1700 行定义的函数 leaveLoopPatch 用于离开循环体时的相关 patch，也就是循环体编译已经结束了，本函数做一些收尾工作。循环肯定免不了向回跳转的指令，即循环体结束处有一条向回跳转到循环体起始处的指令，虚拟机的执行轨迹是用指针 ip 来实现的，因此只要把 ip 指向新地址就实现了转移，对于向回跳转来说，只要把 ip 重新指向循环体开始地址即可。跳转指令的操作数都是偏移量，并不是用绝对地址，比如 OPCODE_JUMP 的操作数就是目标地址运行当前地址的偏移量。不过向回跳转并不是用 OPCODE_JUMP 来实现的，因为跳转偏移量不能为负数，我们用专门的操作码 OPCODE_LOOP 来实现，它的操作数同样也必须为正偏移量，因此其改变执行轨迹的方式是“ip -= 操作数”，此操作数是下一条指令的地址与循环体起始地址的差，即偏移量，因此是正数（除非出错）。第 1702 行获取往回跳转的偏移量，loopBackOffset 是操作码 OPCODE_ LOOP 的操作数，操作数在指令流中占 2 字节，当虚拟机读取到 OPCODE_LOOP 的操作数 loopBack Offset 时，ip 相对 OPCODE_LOOP 来说已经向后偏移了 2 字节，因此跳回的偏移量要加上这 2 个字节。第 1705 行生成 CODE_LOOP 指令，即令指针 ip -= loopBackOffset。由于调用本函数时循环体已经编译结束了，因此第 1708 行回填循环体的退出地址，其中循环的退出地址 curLoop-> exitIndex 是在函数 compileWhileStatment 中占位的。

退出循环体是用 break 实现的，break 实际上也是跳转，因此也需要跳转的目标地址。不过在循环体编译结束前，break 的目标地址是不知道的，我们编译 break 的策略是先把操作码 OPCODE_END 作为 break 的占位符（因为此操作码用不上，不会有二义性），将来待循环体编译结束获得循环体大小后再回填，这个“将来”就是“此时”。在循环中根据不同的情况可能有很多 break。第 1712～1732 行之间就是遍历指令流中所有 OPCODE_END，也就是处理所有的 break，因此循环体中处理完一个 OPCODE_END 后并不退出循环，而是继续遍历。处理很简单，就是用 OPCODE_JUMP 替换 OPCODE_END，然后调用函数 patchPlaceholder 回填 break 的跳转目标地址。然后第 1725 行使 idx+=3，跳过操作数，直指下一个操作码。如果并不是操作码 OPCODE_END，那么为提高遍历速度，在第 1729 行，在指令流中一次性跳过该指令（即操作码及其操作数），通过 getBytesOfOperands 获得指令的操作数大小，左边的 1 字节表示操作码。最后退出循环体，使 cu->curLoop 恢复为当前循环层的外层循环。

以上介绍的全是铺垫，第 1739 行定义的 compileWhileStatment 函数才是本节的主角，它负责编译 while 循环，进入本函数时已经读取了关键字 while，curToken 是 while 后面的符号，语法正确的话应该是“(”。第 1743 行调用 enterLoopSetting 设置循环体起始地址，第 1744～1748 行处理 while 中的（条件表达式）。如果条件失败则跳出循环体，目前尚未编译循环体，因此不知循环体的结束地址，故第 1752 行调用 emitInstrWithPlaceholder 为操作码 OPCODE_JUMP_IF_FALSE 的跳转地址（操作数）用占位符占位。然后调用 compileLoopBody 编译循环体，最后调用 leaveLoopPatch 处理收尾。

第 1764 行把 compileWhileStatment 添加到 compileStatment，即识别到关键字 while 时调用 compileWhileStatment。

6.19.3　编译 return、break 和 continue 语句（本节源码 stepByStep/c6/o）

本节编译 return、break 和 continue 语句，代码如下。

stepByStep/c6/o/compiler/compiler.c

```
...略
410 //丢掉作用域 scopeDepth 之内的局部变量
411 static uint32_t discardLocalVar(CompileUnit* cu, int scopeDepth) {
412     ASSERT(cu->scopeDepth > -1, "upmost scope can´ t exit!");
413     int localIdx = cu->localVarNum - 1;
414
415     //变量作用域大于 scodeDepth 的为其内嵌作用域中的变量,
416     //跳出 scodeDepth 时内层也没用了,要回收其局部变量
417       while (localIdx >= 0 && cu->localVars[localIdx].scopeDepth >= scopeDepth) {
418        if (cu->localVars[localIdx].isUpvalue) {
419            //如果此局部变量是其内层的 upvalue 就将其关闭
420            writeByte(cu, OPCODE_CLOSE_UPVALUE);
421        } else {
422            //否则就弹出该变量回收空间
423            writeByte(cu, OPCODE_POP);
424        }
425        localIdx--;
426     }
427
428     //返回丢掉的局部变量个数
429     return cu->localVarNum - 1 - localIdx;
430 }
...略
1781 //编译 return
1782 inline static void compileReturn(CompileUnit* cu) {
1783     if (PEEK_TOKEN(cu->curParser) == TOKEN_RIGHT_BRACE) { //空返回值
1784        //空 return,NULL 作为返回值
1785        writeOpCode(cu, OPCODE_PUSH_NULL);
1786
1787     } else {  //有返回值
1788        expression(cu, BP_LOWEST);
1789     }
1790     writeOpCode(cu, OPCODE_RETURN);    //将上面栈顶的值返回
1791 }
1792
1793 //编译 break
1794 inline static void compileBreak(CompileUnit* cu) {
1795     if (cu->curLoop == NULL) {
1796        COMPILE_ERROR(cu->curParser, "break should be used inside a loop!");
1797     }
1798
1799     //在退出循环体之前要丢掉循环体内的局部变量
1800     discardLocalVar(cu, cu->curLoop->scopeDepth + 1);
1801
1802     //由于用 OPCODE_END 表示 break 占位，此时无须记录占位符的返回地址
1803     emitInstrWithPlaceholder(cu, OPCODE_END);
1804 }
1805
1806 //编译 continue
```

```
1807 inline static void compileContinue(CompileUnit* cu) {
1808     if (cu->curLoop == NULL) {
1809         COMPILE_ERROR(cu->parser, "continue should be used inside a loop!");
1810     }
1811
1812     //回收本作用域中局部变量在栈中的空间，+1 是指循环体(包括循环条件)的作用域
1813     //不能在 cu->localVars 数组中去掉，
1814     //否则在 continue 语句后面若引用了前面的变量则提示找不到
1815     discardLocalVar(cu, cu->curLoop->scopeDepth + 1);
1816
1817     int loopBackOffset = cu->fn->instrStream.count - cu->curLoop->
        condStartIndex + 2;
1818
1819     //生成向回跳转的 CODE_LOOP 指令，即使 ip -= loopBackOffset
1820     writeOpCodeShortOperand(cu, OPCODE_LOOP, loopBackOffset);
1821 }
1822
1823 //编译语句(即程序中与声明、定义无关的表示“动作”的代码)
1824 static void compileStatment(CompileUnit* cu) {
1825     if (matchToken(cu->curParser, TOKEN_IF)) {
1826         compileIfStatment(cu);
1827     } else if (matchToken(cu->curParser, TOKEN_WHILE)) {
1828         compileWhileStatment(cu);
1829     } else if (matchToken(cu->curParser, TOKEN_RETURN)) {
1830         compileReturn(cu);
1831     } else if (matchToken(cu->curParser, TOKEN_BREAK)) {
1832         compileBreak(cu);
1833     } else if (matchToken(cu->curParser, TOKEN_CONTINUE)) {
1834         compileContinue(cu);
1835     }
1836 }
…略
```

正式开始之前先介绍第 411 行的函数 discardLocalVar，它用于丢掉作用域 scopeDepth 之内的局部变量，即找出作用域大于 scopeDepth 的变量并丢弃，最后返回丢掉的局部变量的个数。X 层中的局部变量对于嵌套的 X+1 层来说，也许是 upvalue，所以“丢弃”在此的意思是把局部变量弹出或者把 upvalue 关闭。第 412 行通过 ASSERT 判断编译器编译是否正处于模块作用域中，如果是就报错，因为模块作用域是顶层作用域不可退出。第 413～426 行从后面往前找局部变量，因为前面说过了，作用域最深的局部变量在数组 cu->localVars 的后面。第 417 行的 while 循环找出此作用域以内作用域中的局部变量，因为跳出作用域时，其内部作用域也就失效了，因此其内部作用域中的局部变量也访问不到了，应该被丢弃。第 418 行对于 upvalue，生成操作码 OPCODE_CLOSE_UPVALUE 关闭此 upvalue，对于一般的局部变量，那么就在第 423 行生成操作码 OPCODE_POP 弹栈。注意，这里用 writeByte 代替 writeOpCode，目的是不想影响 compileUnit->fn->maxStackSlotUsedNum，因为局部变量是存储在运行时栈中，退出作用域后函数的运行时栈并未回收，也就是说变量仍然在栈中，不可影响到实际栈大小（函数运行时栈的回收由虚拟机在运行时处理）。最后第 429 行返回总共丢弃的变量个数。

第 1782 行定义的函数 compileReturn 用于编译 return 语句，进入本函数时已经读取了关键字

return，curToken 是关键字“return”之后的符号。return 有两种形式，一种是“return 返回值”，从函数返回并把返回值传递给主调方；另一种是“return”，无返回值，只用于从函数返回。这里判断第二种形式的方法比较粗糙，如果 return 后面是“}”，那么我们就认为此 return 后面没有返回值（这样处理是有问题的，如果 return 空语句并未出现在代码块结束处，比如位于代码块中间，那么这种判断就是错的，目前先这样，以后再修正），即只是一句简单的 return，那么就在第 1785 行压入 null 作为默认的返回值，这也是为了和第 1788 行的 expression 把返回值留在栈顶，即 return 有返回值时保持相同的栈结构。

第 1794 行定义的函数 compileBreak 用于编译 break 语句，break 用于退出循环体。调用本函数时已经读入了关键字 break，curToken 是 break 后面的符号。第 1795 行确认 break 只能用在循环中。在退出循环体之前，第 1800 行调用 discardLocalVar 丢弃循环体内的局部变量，注意，此处用当前循环的作用域+1 做参数，即丢弃比此作用域更深作用域内的变量。break 语句只位于循环体中，用于跳出循环体外，因此就需要把循环体结束地址的下一地址（表示循环体外）作为操作数。在编译 break 语句的时候尚未编译完循环体，因此不知道循环体的结束地址，所以不知道 break 的目标地址，就像编译 if 语句一样，在不知道目标地址的情况下需要将目标地址临时用个占位符替代，待将来编译好循环体后再回填，不过与 if 不同的是，在循环体中出现的 break 未必是一句，用户可能根据不同的条件执行多个 break，因此要为所有 break 占位，将来一并回填。如果将本循环中所有 break 占位符的索引收集起来，那么就需要额外的结构来保存，因此不如这样，既然占位时肯定要传个操作码，那我们就用没用的 OPCODE_END 来表示 break 语句，将来编译完循环体后回填时，在循环体的指令流中遍历操作码 OPCODE_END 就行了。提醒一下，不要觉得各个 break 的操作数不同，同一个循环体中所有 break 的操作数都是该循环体结束地址的下一个地址，因此我们才可以在编译完循环体后统一回填。

第 1807 行的函数 compileContinue 用于编译 continue 语句，进入本函数时已经读取了关键字 continue。continue 语句用于结束本次循环，进入下一轮循环，即不执行循环体中 continue 语句后面的代码，因此 continue 语句只能用在循环里。在我们的语言中，循环语句就是指 while 和 for。函数前 3 行判断 continue 当前是否处在循环语句的环境中，如果 cu->curLoop 为 NULL，说明未在循环中，因此报错。

continue 的原理很简单，此语句的功能是用操作码 OPCODE_LOOP 实现的，此操作码可使程序计数器 ip 往回跳转，因此在原理上我们只要跳到循环体的循环条件处就可以实现 continue 了，但要做好“善后”工作，循环体本身是个代码块，代码块就是个作用域，因此循环体内部可以有很多局部变量，而局部变量就涉及栈空间。比如可以在循环体中用 var 定义局部变量，这样编译器会在栈中为局部变量分配空间以存储其“值”（提醒：局部变量“名”是存储在 compileUnit->localVars[局部变量索引]中）。再者，我们是基于栈的虚拟机，很多指令的操作数都是位于栈顶，比如循环条件就要加载到栈顶，这也会占用栈的空间。在循环结构中不断循环执行的不仅是循环体，还包括循环条件，这是必然的，因为只有不断更新循环条件，才能确保循环的退出时机。不过好在循环条件所占的栈空间会由操作码 JUMP_IF_FALSE 来回收，第 7 章我们就会知道，这是通过 POP 来实现的。如果循环体中包含定义变量的代码，如果指令未优化的话，那么循环执行时，就会循环创建变量，也就是不断地为局部变量在栈中分配空间，这会导致栈溢出。看如下两个死循环（代码没有实际意义）。

```
while(true) {
   var a = 1
   continue
}
```

```
while(true) {
   continue
   var a = 1
}
```

对于左框代码，在死循环中会不断为变量 a 分配空间：编译器会用 LOAD_CONSTANT 1 把常量 1 加载到栈顶，此时栈顶便是变量 a 的空间，如果仅仅是变量声明即 var a，那么编译器会生成 PUSH_NULL 以使变量 a 的值为 null，总之都会为变量在栈中分配一个 slot，那么死循环分配一定会栈溢出。对于右框中的代码则不会溢出，因为 var 语句是在 continue 的后面，虚拟机执行 continue 对应的指令（OPCODE_LOOP 及其操作数）后会跳向循环条件处的指令（在此为 PUSH_TRUE），即变量 a 对应的指令（LOAD_CONSTANT 1）没机会执行，因此不会分配空间，更不会溢出。也许有读者会问，如果循环体中没有 continue 语句，那么这个死循环也会因为不断分配局部变量 a 而耗尽栈空间吗？当然不会，因为循环体也是个代码块，代码块就是个作用域，我们在调用 compileBlock 编译代码块时的前后会分别调用 enterScope 和 leaveScope 为代码块创建作用域及出离作用域。在出离循环体时，我们会回收循环体中局部变量的空间，这是由 leaveScope 中调用 discardLocalVar 函数完成的。因此是在循环为局部变量分配空间和释放空间，所以栈并不会溢出。前面说过了，这是在未优化的情况下才会如此“笨拙”，遗憾的是我们目前还没有中间码优化的阶段，希望以后会加上。不过优化思路可以“畅想”一下，可以把变量 a 的空间固定下来，不要每次都重新分配，比如把循环体中的局部变量往上提升一层作为直接外层的局部变量来编译（为了确保不与上层局部变量名冲突，要把变量名转换映射为不冲突的变量名，比如可以把作用域链作为变量名的前缀），或者直接把局部变量空间一次性分配好，直到循环结束后再一次性回收。

第 1815 行调用 discardLocalVar 回收循环体内局部变量的空间，然后在后面生成操作码 OPCODE_LOOP 以往回跳偏移量 loopBackOffset 以执行流回到条件判断处。

最后在函数 compileStatment 中分别用这两个函数处理关键字 return、break 和 continue 的编译。

6.19.4　编译 for 循环语句（本节源码 stepByStep/c6/p）

本节编译 for 循环。for 循环语法格式是 for i (sequence) {循环体}。

stepByStep/c6/p/compiler/compiler.c

```
...略
 432 //进入新作用域
 433 static void enterScope(CompileUnit* cu) {
 434       //进入内嵌作用域
 435       cu->scopeDepth++;
 436 }
 437
 438 //退出作用域
 439 static void leaveScope(CompileUnit* cu) {
 440       //对于非模块编译单元,丢弃局部变量
 441       if (cu->enclosingUnit != NULL) {
 442           //出作用域后丢弃本作用域以内的局部变量
 443           uint32_t discardNum = discardLocalVar(cu, cu->scopeDepth);
 444           cu->localVarNum -= discardNum;
 445           cu->stackSlotNum -= discardNum;
 446       }
 447
 448       //回到上一层作用域
 449       cu->scopeDepth--;
 450 }
...略
1802 //编译 for 循环,如 for i (sequence) {循环体}
```

```
1803 static void compileForStatment(CompileUnit* cu) {
1804 //其中 sequence 是可迭代的序列
1805
1806 //for 循环会按照 while 循环的逻辑编译
1807 //   for i (sequence) {
1808 //       System.print(i)
1809 //   }
1810 //
1811 //在内部会变成:
1812 //   var seq = sequence
1813 //   var iter
1814 //   while iter = seq.iterate(iter) {
1815 //      var i = seq.iteratorValue(iter)
1816 //      System.print(i)
1817 //   }
1818
1819     //为局部变量 seq 和 iter 创建作用域
1820     enterScope(cu);
1821
1822     //读取循环变量的名字,如 for i (sequence)中 i
1823     consumeCurToken(cu->curParser,
1824           TOKEN_ID, "expect variable after for!");
1825     const char* loopVarName = cu->curParser->preToken.start;
1826     uint32_t loopVarLen = cu->curParser->preToken.length;
1827
1828     consumeCurToken(cu->curParser,
1829           TOKEN_LEFT_PAREN, "expect '(' befor sequence!");
1830
1831     //编译迭代序列
1832     expression(cu, BP_LOWEST);
1833     consumeCurToken(cu->curParser, TOKEN_RIGHT_PAREN, "expect ')' after
        sequence!");
1834     //申请一个局部变量 seq 来存储序列对象,
1835     //其值就是上面 expression 存储到栈中的结果
1836     uint32_t seqSlot = addLocalVar(cu, "seq ", 4);
1837
1838     writeOpCode(cu, OPCODE_PUSH_NULL);
1839     //分配及初始化 iter,其值就是上面加载到栈中的 NULL
1840     uint32_t iterSlot = addLocalVar(cu, "iter ", 5);
1841
1842     Loop loop;
1843     enterLoopSetting(cu, &loop);
1844
1845     //为调用 seq.iterate(iter)做准备
1846     //1. 先压入序列对象 seq, 即 seq.iterate(iter)中的 seq
1847     writeOpCodeByteOperand(cu, OPCODE_LOAD_LOCAL_VAR, seqSlot);
1848     //2. 再压入参数 iter,即 seq.iterate(iter)中的 iter
1849     writeOpCodeByteOperand(cu, OPCODE_LOAD_LOCAL_VAR, iterSlot);
1850     //3. 调用 seq.iterate(iter)
1851     emitCall(cu, 1, "iterate(_)", 10);
```

```
1852
1853        //seq.iterate(iter)把结果(下一个迭代器)存储到
1854        //args[0](即栈顶),现在将其同步到变量 iter
1855        writeOpCodeByteOperand(cu, OPCODE_STORE_LOCAL_VAR, iterSlot);
1856
1857        //如果条件失败则跳出循环体,目前不知道循环体的结束地址,
1858        //先写入占位符
1859                    loop.exitIndex= emitInstrWithPlaceholder(cu, OPCODE_JUMP_IF_
FALSE);
1860
1861        //调用 seq.iteratorValue(iter)以获取值
1862        //1. 为调用 seq.iteratorValue(iter)压入参数 seq
1863        writeOpCodeByteOperand(cu, OPCODE_LOAD_LOCAL_VAR, seqSlot);
1864        //2. 为调用 seq.iteratorValue(iter)压入参数 iter
1865        writeOpCodeByteOperand(cu, OPCODE_LOAD_LOCAL_VAR, iterSlot);
1866        //3. 调用 seq.iteratorValue(iter)
1867        emitCall(cu, 1, "iteratorValue(_)", 16);
1868
1869        //为循环变量 i 创建作用域
1870        enterScope(cu);
1871        //seq.iteratorValue(iter)已经把结果存储到栈顶,
1872        //添加循环变量为局部变量,其值在栈顶
1873        addLocalVar(cu, loopVarName, loopVarLen);
1874
1875        //编译循环体
1876        compileLoopBody(cu);
1877
1878        leaveScope(cu); //离开循环变量 i 的作用域
1879
1880        leaveLoopPatch(cu);
1881
1882        leaveScope(cu); //离开变量 seq 和 iter 的作用域
1883 }
...略
1928 //编译语句(即程序中与声明、定义无关的表示“动作”的代码)
1929 static void compileStatment(CompileUnit* cu) {
1930        if (matchToken(cu->curParser, TOKEN_IF)) {
1931            compileIfStatment(cu);
1932        } else if (matchToken(cu->curParser, TOKEN_WHILE)) {
1933            compileWhileStatment(cu);
1934        } else if (matchToken(cu->curParser, TOKEN_FOR)) {
1935            compileForStatment(cu);
1936        } else if (matchToken(cu->curParser, TOKEN_RETURN)) {
1937            compileReturn(cu);
1938        } else if (matchToken(cu->curParser, TOKEN_BREAK)) {
1939            compileBreak(cu);
1940        } else if (matchToken(cu->curParser, TOKEN_CONTINUE)) {
1941            compileContinue(cu);
1942        }
1943 }
...略
```

第 433 行定义的 enterScope 用于进入一个新的内层作用域，作用域是用 compileUnite->scopeDepth 记录的，因此本函数就是使 scopeDepth++。

第 439 行定义的 leaveScope 用于退出作用域。模块和方法有编译单元，类没有编译单元，cu->enclosingUnit == NULL 时表示 cu 是模块编译单元，同时也表示正在编译类定义（不在方法的编译单元中）。退出作用域时要回收该作用域内的局部变量，类的静态域是作为模块的局部变量，如果不加以限制，退出类定义时会回收类中静态域，因此第 441 行表示只回收非模块编译单元中的局部变量，这样确保类中静态域得以保留。第 449 行使 scopeDepth--以回到父作用域。

第 1803 行的 compileForStatment 用于编译 for 循环，for 循环语法是 for i (sequence) {循环体}，其中 sequence 是可迭代的序列。什么是可迭代序列呢？就是指拥有 iterate 方法的对象，只要对象具有 iterate 方法我们便认为其可迭代，也就是说，只要提供给 for 的循环对象具有 iterate 方法即可，比如 ObjString、ObjRange、ObjList。第 1806～1817 行是编译 for 循环的原理，把 for 循环按照 while 循环来处理，原注释如下。

for 循环会按照 while 循环的逻辑编译

```
for i (sequence) {
    System.print(i)
}
```

在内部会变成:

```
var seq = sequence
var iter
while iter = seq.iterate(iter) {
   var i = seq.iteratorValue(iter)
   System.print(i)
}
```

编译 for 的步骤如下。

（1）对序列 sequence 生成可供迭代的对象 seq。

（2）调用 seq 对象的 iterate(iter)方法生成迭代器 iter，返回值通常是参数 iter 加 1，这取决于 iterate 的实现。

（3）调用 seq 对象的 iteratorValue(iter)方法获取迭代器 iter 对应的值 i。

（4）重复执行步骤（2）～（4），直到第（2）步返回值即迭代器为空。

在后面，我们会将所有可迭代序列对象添加 iterate 和 iteratorValue 方法，目前只要理解这个模式就行。把以上步骤结合代码注释来理解。

第 1820 行调用 enterScope 为内部变量创建独立作用域，后面会看到这两个内部变量是 seq 和 iter。第 1823～1833 行处理 for i (sequence){}中的 i (sequence)。第 1832 行的 expression 最终是把序列对象存储到栈顶，第 1836 行申请局部变量 seq 来关联该序列对象，为的是以后可用局部变量 seq 来使用此序列对象。为了避免与用户变量冲突，我们在内部变量名后面加个空格，即表示 seq 的变量名是 seq。另一个内部变量是 iter，第 1838～1840 行创建及初始化 iter 为 null，即表示初始的 iter 为空。迭代器就是整数索引，每调用 seq 对象的 iterate(iter)方法，返回值就是 iter+1。因此第一次调用 seq 对象的 iterate 方法时，参数 iter 肯定是 null，这样 seq. iterate(null)返回值为 0，即返回可迭代序列的第 0 个索引。提醒，变量名 iter 结尾也有个空格。第 1843 行调用 enterLoopSetting 进行编译循环体前的设置。我们通过调用序列对象的.iterate 方法去移进迭代器，下面为调用.iterate 自左向右的顺序传递参数，参数都在栈中，即生成把参数压栈的指令，消息的接收者即对象始终要最先压入使其成为 args[0]。第 1846～1849 行是为调用 seq.iterate(iter)传递参数 seq 和 iter，参数

需要按照从左到右的顺序压栈，即先压入 seq，再压入 iter。

第 1851 行是生成调用 seq.iterate(iter)的指令。当虚拟机执行完调用 seq.iterate(iter)的指令后，栈顶会是下一个迭代器（的值）。第 1855 行生成操作码 OPCODE_STORE_LOCAL_VAR，把迭代器同步到变量 iter，也就是实现了迭代器更新。第 1859 行表示如果条件失败则跳出循环体，目前尚未编译循环体，不知循环体的结束地址，先把操作码 OPCODE_JUMP_IF_FALSE 的跳转地址写入占位符，以后回填。接下来是为调用 seq.iteratorValue(iter)传入参数 seq 和 iter。第 1867 行是生成调用 seq.iteratorValue(iter)的指令。在虚拟机执行生成的指令后，栈顶处就是最新的 iter 值。

接下来要处理循环变量，循环变量就是 for i (sequence)中的 i，调用 enterScope 为其创建作用域。下面将栈顶结果即 seq.iteratorValue(iter)的返回值存储到循环变量，第 1873 行用循环变量名为局部变量，用栈顶的值为其值，也就是实现了序列的迭代值赋给了循环变量。接下来调用 compileLoopBody 编译循环体，之后调用 leaveScope 离开循环变量的作用域并丢弃相关局部变量，然后调用 leaveLoopPatch 进行编译后的收尾工作，最后调用 leaveScope 离开内部变量 seq 和 iter 的作用域并丢弃变量。

接下来在函数 compileStatment 中添加 compileForStatment，用来编译关键字 for。

6.19.5　编译代码块及单一语句（本节源码 stepByStep/c6/q）

本节编译代码块和单一语句。代码块就是由大括号对儿括起的代码，大括号对儿{}所起的作用就是单独的作用域。单一语句很简单，就是一行代码。

stepByStep/c6/q/compiler/compiler.c

```
1927 //编译语句(即程序中与声明、定义无关的表示“动作”的代码)
1928 static void compileStatment(CompileUnit* cu) {
1929     if (matchToken(cu->curParser, TOKEN_IF)) {
1930         compileIfStatment(cu);
1931     } else if (matchToken(cu->curParser, TOKEN_WHILE)) {
1932         compileWhileStatment(cu);
1933     } else if (matchToken(cu->curParser, TOKEN_FOR)) {
1934         compileForStatment(cu);
1935     } else if (matchToken(cu->curParser, TOKEN_RETURN)) {
1936         compileReturn(cu);
1937     } else if (matchToken(cu->curParser, TOKEN_BREAK)) {
1938         compileBreak(cu);
1939     } else if (matchToken(cu->curParser, TOKEN_CONTINUE)) {
1940         compileContinue(cu);
1941     } else if (matchToken(cu->curParser, TOKEN_LEFT_BRACE)) {
1942         //代码块有单独的作用域
1943         enterScope(cu);
1944         compileBlock(cu);
1945         leaveScope(cu);
1946     } else {
1947         //若不是以上的语法结构则是单一表达式
1948         expression(cu, BP_LOWEST);
1949
1950         //表达式的结果不重要,弹出栈顶结果
1951         writeOpCode(cu, OPCODE_POP);
1952     }
1953 }
```

本节比较轻松，就是往函数 compileStatement 中增加一些代码，第 1925～1929 行用于处理代码块，代码块的识别是以左大括号“{”为标记。代码块既然是用大括号包起来，说明这是单独的作用域，因此在调用 compileBlock 编译代码块之前先调用 enterScope 创建新的作用域，编译代码块之后调用 leaveScope 退出作用域。

下面第 1947～1951 行是编译单一表达式，单一表达式就是单一的语句，比如 1+2 就是单一的语句。直接调用 expression 就可以编译一行语句。但是下面为什么要生成操作码 OPCODE_POP 呢？我们知道，expression 是负责生成计算表达式的一系列指令，在虚拟机执行这一系列指令后，栈顶就是表达式的最终结果。不过，此表达式的结果并不重要，因此它所在的环境并不要求保留结果，比如条件表达式中的条件是用 expression 单独解析的，表达式的结果作为条件判断，因此结果很重要，而此 compileStatement 并不用于条件表达式中的条件部分，本函数只是处理语句，比如对于 1+2 来说，结果是 3，它位于栈顶，但是 3 并不被后续表达式使用，因此 3 就可以从栈中 pop 出去。再比如 a=1+2，右边的结果依然是 3，3 要同步到变量 a 中，但是同步到 a 中的只是 3 的副本，也就是说只是把数值 3 复制到变量 a 所在运行时栈相应的 slot 中。栈顶中的 3 还在，因此栈顶中的结果依然要被弹出。

也许读者现在对单一语句编译流程感到困惑，举个例子也许会清楚一些。下面 t.sp 中共两行代码，按照上面的编译逻辑，会按照第 1947～1951 行的逻辑处理，因此会被处理两次，也就是函数 compileStatement 会被调用两次，第一次先处理 System.print(1)，第二次处理 System.print(2)。

```
[work@work spr]$ cat -n /tmp/t.sp
     1System.print(1)
     2System.print(2)
[work@work spr]$
```

6.20 编译类定义（本节源码 stepByStep/c6/r）

6.20.1 方法的声明与定义

本节完成类中方法的声明与定义。

先看下声明和定义的区别，“声明”不包括“值”，比如 var a 就声明了变量 a；“定义”是指声明的同时赋值，即名称和值绑定才叫定义。对于方法来说也涉及声明和定义，方法由方法名和方法体两部分构成，方法名就是方法的标识，方法体相当于方法的值，方法名存储在虚拟机 vm->allMethodNames，方法体是存储方法所属类 class->methods，因此声明方法就是往 vm->allMethodNames 中注册方法名，而定义方法就是把方法体写入 class->methods。

stepByStep/c6/r/compiler/compiler.c

```
...略
1065 //生成存储模块变量的指令
1066 static void emitStoreModuleVar(CompileUnit* cu, int index) {
1067       //把栈顶数据存储到 moduleVarValue[index]
1068       writeOpCodeShortOperand(cu, OPCODE_STORE_MODULE_VAR, index);
1069       writeOpCode(cu, OPCODE_POP);  //弹出栈顶数据
1070 }
...略
1962 //声明方法
1963 static int declareMethod(CompileUnit* cu, char* signStr, uint32_t length) {
1964       //确保方法被录入到 vm->allMethodNames
```

```
1965      int index = ensureSymbolExist(cu->curParser->vm,
1966            &cu->curParser->vm->allMethodNames, signStr, length);
1967
1968      //下面判断方法是否已定义，即排除重定义
1969      IntBuffer* methods = cu->enclosingClassBK->inStatic ?
1970            &cu->enclosingClassBK->staticMethods :
1971            &cu->enclosingClassBK->instantMethods;
1972      uint32_t idx = 0;
1973      while (idx < methods->count) {
1974         if (methods->datas[idx] == index) {
1975            COMPILE_ERROR(cu->curParser, "repeat define method %s in class %s!",
1976               signStr, cu->enclosingClassBK->name->value.start);
1977         }
1978         idx++;
1979      }
1980
1981      //若是新定义就加入,这里并不是注册新方法,
1982      //而是用索引来记录已经定义过的方法,用于以后排重
1983      IntBufferAdd(cu->curParser->vm, methods, index);
1984      return index;
1985 }
1986
1987 //将方法 methodIndex 指代的方法装入 classVar 指代的 class.methods 中
1988 static void defineMethod(CompileUnit* cu,
1989         Variable classVar, bool isStatic, int methodIndex) {
1990 //1. 待绑定的方法在调用本函数之前已经放到栈顶了
1991
1992 //2. 将方法所属的类加载到栈顶
1993      emitLoadVariable(cu, classVar);
1994
1995 //3. 生成 OPCODE_STATIC_METHOD 或 OPCODE_INSTANCE_METHOD
1996      //在运行时绑定
1997      OpCode opCode = isStatic ?
1998        OPCODE_STATIC_METHOD : OPCODE_INSTANCE_METHOD;
1999      writeOpCodeShortOperand(cu, opCode, methodIndex);
2000 }
…略
```

先介绍函数 emitStoreModuleVar，它用来生成存储模块变量的操作码，也就是 OPCODE_STORE_MODULE_VAR。以后在实现虚拟机时你会发现，存储模块变量的本质是根据模块变量的索引 index，把栈顶数据存储到模块变量的容身之所，即 module->moduleVarValue[index]。栈顶数据写入模块变量后就没用了，因此生成操作码 OPCODE_POP 弹出栈顶数据。

其实前面所说的方法声明和定义就是方法的管理策略，曾经也和大家大体介绍过，编译器在编译阶段查找方法时是从 vm->allMethodNames 中查找，获得其索引，用此索引生成调用指令，虚拟机在运行阶段用索引在 class.methods 中查找方法并执行该方法。vm->allMethodNames 中的方法名是谁装进去的呢？就是第 1963 行的函数 declareMethod 完成的，它用于声明方法，也就是往 vm->allMethodNames 写入方法名供以后查找。第 1965 行就是调用 ensureSymbolExist 方法确保方法名写入 vm->allMethodNames，此函数只确保方法名存在，如果之前已经插入就直接返回方法名

的索引，如果不存在则插入，然后返回索引。无论如何 ensureSymbolExist 都会返回索引，从索引上无法看出此方法是否重定义，为了跟踪此情况，我们用了 ClassBookKeep 结构中的两个缓冲区记录方法的定义情况，对于实例方法则用 ClassBookKeep->instantMethods 跟踪，对于类方法则用 ClassBookKeep->staticMethods 跟踪。这里所谓的跟踪就是把方法在 vm->allMethodNames 中的索引写入到这两个缓冲区中，编译方法定义时用方法的索引在这两个缓冲区中检查，如果已存在则表示重定义，因此这两个缓冲区只是用于统计类中方法的定义情况，仅用于排重，不要误认为这就是定义方法，我们定义方法是用后面的 defineMethod。第 1969～1983 行就是按照以上逻辑判断方法是否重定义。最后第 1984 行返回方法的索引。

第 1988 行定义的 defineMethod 用于定义方法，本质上是将方法索引 methodIndex 指代的方法装入 classVar 指代的类的 class.methods[methodIndex]中。待定义（绑定）的方法在调用本函数之前已经放到栈顶了，这是由函数 endCompileUnit 中生成的 CODE_CLOSURE 指令放到栈中，还是那句话，将来实现虚拟机时大家就清楚了（语义分析就是和指令的执行相配合的）。此时栈顶是待定义的方法，第 1993 行调用 emitLoadVariable 函数把方法所属的类加载到栈顶，此时待定义的方法便在次栈顶了，最后根据是否是静态方法生成相应的操作码 OPCODE_STATIC_METHOD 或 OPCODE_INSTANCE_METHOD，虚拟机在执行这两个操作码时会把栈顶和次栈顶中的类和方法取出来，从指令流中取出方法的索引 methodIndex，然后把方法存储到类的 methods[methodIndex]，这就是定义。

6.20.2 构造函数与创建对象

实例对象就是我们在调用一个类的构造方法后所获得的返回值，一般的构造方法是类名，或者可以自由指定，或者是 new 等，我们的构造方法就是 new，如图 6-5 所示。

```
class Foo{
    var bar
    new(arg){
        bar=arg
    }
}
var obj=Foo.new(9)
```

图 6-5

类中方法分为两类，实例方法和类方法，实例方法属于具体的实例对象，可以理解为每个实例对象有自己单独的实例方法。类方法也称为静态方法，在我们语言中是用关键字 static 修饰，静态方法被所有实例对象共享，调用方式是“类名.静态方法”。说明一下，在我们语言中实例对象不允许调用静态方法，静态方法也不允许访问实例域。观察图 6-5，生成一个对象是调用“类名.new()”，这说明 new 虽然没有用关键字 static 修饰，但它也被当作静态方法处理，这理解起来也很容易，毕竟在没有实例对象的时候也谈不上调用实例方法，必须用一个公用的方法即静态方法才能创建对象。构造函数是用来创建实例对象的，在构造函数中难免会涉及与实例对象相关的代码，比如图 6-5 中的 new 方法用形参 arg 初始化实例域 bar，但是实例域属于实例对象，前面说过静态方法是不允许访问实例域的，这说明构造函数是个特殊的函数，具有双重身份，“表面”上同时是静态方法和实例方法，这又是怎么回事呢？先看代码。

stepByStep/c6/r/compiler/compiler.c

```
...略
2002 //分两步创建实例,constructorIndex 是构造函数的索引
2003 static void emitCreateInstance(CompileUnit* cu,
```

```
2004        Signature* sign, uint32_t constructorIndex) {
2005    CompileUnit methodCU;
2006    initCompileUnit(cu->curParser, &methodCU, cu, true);
2007
2008 //1. 生成 OPCODE_CONSTRUCT 指令,该指令生成新实例存储到 stack[0]
2009    writeOpCode(&methodCU, OPCODE_CONSTRUCT);
2010
2011 //2. 生成 OPCODE_CALLx 指令,该指令调用新实例的构造函数
2012    writeOpCodeShortOperand(&methodCU,
2013          (OpCode)(OPCODE_CALL0 + sign->argNum), constructorIndex);
2014
2015 //生成 return 指令,将栈顶中的实例返回
2016    writeOpCode(&methodCU, OPCODE_RETURN);
2017
2018 #if DEBUG
2019    endCompileUnit(&methodCU, "", 0);
2020 #else
2021    endCompileUnit(&methodCU);
2022 #endif
2023 }
...略
```

方法 emitCreateInstance 表面上是创建静态方法 new，实际上就是用于创建实例对象，其中形参 constructorIndex 是构造函数的索引。函数开头定义了 methodCU，它专用于存储创建对象的指令，调用了 initCompileUnit 使其作为 cu 的内层。创建对象免不了要用到操作码 OPCODE_CONSTRUCT，第 1993 行生成此操作码，虚拟机在执行此操作码后会生成新实例，然后把实例存储到运行时栈第 0 个 slot 中，即 stackStart[0]。第 2012 行生成操作码 OPCODE_CALL0 去调用实例对象的 new 方法。第 2016 行生成操作码 OPCODE_RETURN 返回实例方法 new 返回的对象。最后调用 endCompileUnit 生成静态方法 new 的闭包。

下面说下构造函数生成对象的流程，编译器是这样处理构造函数的。

（1）编译类定义时生成操作码 OPCODE_CREATE_CLASS，此操作码会在运行时栈的第 0 个 slot，即 stackStart[0]中创建类，注意此时 stackStart[0]中是 class 结构，不是实例对象。后面我们在 compileClassDefinition 函数中会看到此操作码的生成过程。

（2）new 方法是作为实例方法步骤（3）绑定到实例对象所属类的 methods。编译 new 方法体时（是用 compileBody 编译所有方法的方法体，包括构造方法），compileBody 针对构造函数在方法体的指令流中额外添加了两条指令：OPCODE_LOAD_LOCAL_VAR,0 和 OPCODE_RETURN，目的是把步骤（4）中创建的对象返回。

（3）方法编译后都是个闭包，先用 defineMethod 把 new 方法闭包绑定到对象实例，即对象所属类的 methods 中。切记，此时的 new 方法属于某个对象实例，即是实例方法，不是类方法，因为类方法是绑定在对象所属类的类（即元类）中，步骤（4）中创建的是 new 的静态方法。

（4）emitCreateInstance 中为静态 new 方法创建了闭包，闭包指令流中包括 OPCODE_CONSTRUCT 和 OPCODE_CALL。OPCODE_CONSTRUCT 创建对象，然后 OPCODE_CALL 调用步骤（3）中的实例方法 new，此 new 可返回由 OPCODE_CONSTRUCT 放在 stackStart[0]中的对象实例，这是由步骤（2）中额外添加的两条指令完成的。

（5）把步骤（4）中创建的闭包通过 defineMethod 绑定在类的元类中的 methods。这使得构造函数 new 成为类方法，从而保证创建对象时“类名.new()”可以成功。

构造函数的编译是以（1）～（5）的顺序，而构造函数的运行入口是（5）。

总结一下，“类名.new()”是调用的类的静态方法 new，此 new 并不是脚本中写的 new（脚本中的 new 是实例方法，由步骤（2）和（3）生成），而是步骤（4）和（5）中创建的静态 new，此方法包括操作码 OPCODE_CONSTRUCT 和 OPCODE_CALL，由 OPCODE_CALL 去调用步骤（3）中创建的实例 new 方法，该方法中已经由 compileBody 添加了额外的两条指令 OPCODE_LOAD_LOCAL_VAR,0 和 OPCODE_RETURN，这样以返回步骤（4）中由操作码 OPCODE_CONSTRUCT 创建的实例对象。构造函数调用过程如图 6-6 所示。

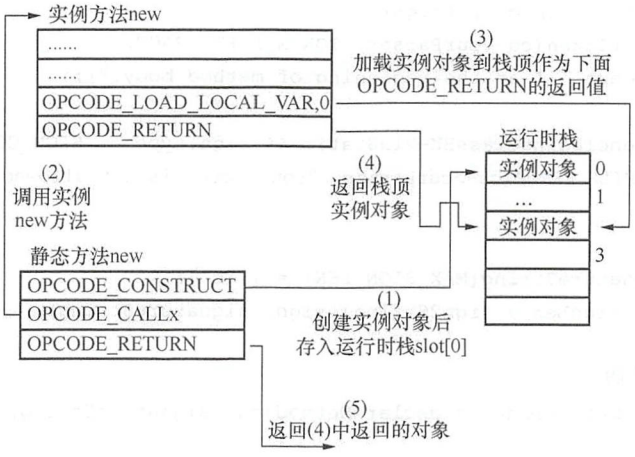


图 6-6

将来大家通读完整代码吧，结合虚拟机来理解就容易了，总之不看代码的话很难理解内部的动作。

6.20.3 编译方法

本节完成方法的编译。

stepByStep/c6/r/compiler/compiler.c

```
...略
2025 //编译方法定义,isStatic 表示是否在编译静态方法
2026 static void compileMethod(CompileUnit* cu, Variable classVar, bool isStatic) {
2027     //inStatic 表示是否为静态方法的编译单元
2028     cu->enclosingClassBK->inStatic = isStatic;
2029     methodSignatureFn methodSign =
2030        Rules[cu->curParser->curToken.type].methodSign;
2031     if (methodSign == NULL) {
2032             COMPILE_ERROR(cu->curParser, "method need signature function!");
2033     }
2034
2035     Signature sign;
2036     // curToken 是方法名
2037     sign.name = cu->curParser->curToken.start;
2038     sign.length = cu->curParser->curToken.length;
2039     sign.argNum = 0;
2040
2041     cu->enclosingClassBK->signature = &sign;
```

```
2042    getNextToken(cu->curParser);
2043
2044    //为了将函数或方法自己的指令流和局部变量单独存储,
2045    //每个函数或方法都有自己的 CompileUnit
2046    CompileUnit methodCU;
2047    //编译一个方法,因此形参 isMethod 为 true
2048    initCompileUnit(cu->curParser, &methodCU, cu, true);
2049
2050    //构造签名
2051    methodSign(&methodCU, &sign);
2052    consumeCurToken(cu->curParser, TOKEN_LEFT_BRACE,
2053         "expect '{' at the beginning of method body.");
2054
2055    if (cu->enclosingClassBK->inStatic && sign.type == SIGN_CONSTRUCT) {
2056         COMPILE_ERROR(cu->curParser, "constuctor is not allowed to be static!");
2057    }
2058
2059    char signatureString[MAX_SIGN_LEN] = {'\0'};
2060    uint32_t signLen = sign2String(&sign, signatureString);
2061
2062    //将方法声明
2063    uint32_t methodIndex = declareMethod(cu, signatureString, signLen);
2064
2065    //编译方法体指令流到方法自己的编译单元 methodCU
2066    compileBody(&methodCU, sign.type == SIGN_CONSTRUCT);
2067
2068 #if DEBUG
2069    //结束编译并创建方法闭包
2070    endCompileUnit(&methodCU, signatureString, signLen);
2071 #else
2072    //结束编译并创建方法闭包
2073    endCompileUnit(&methodCU);
2074 #endif
2075
2076    //定义方法:将上面创建的方法闭包绑定到类
2077       defineMethod(cu, classVar, cu->enclosingClassBK->inStatic, methodIndex);
2078
2079    if (sign.type == SIGN_CONSTRUCT) {
2080         sign.type = SIGN_METHOD;
2081         char signatureString[MAX_SIGN_LEN] = {'\0'};
2082         uint32_t signLen = sign2String(&sign, signatureString);
2083
2084         uint32_t constructorIndex = ensureSymbolExist(cu->curParser->vm,
2085                  &cu->curParser->vm->allMethodNames, signatureString, signLen);
2086
2087         emitCreateInstance(cu, &sign, methodIndex);
2088
2089         //构造函数是静态方法,即类方法
2090         defineMethod(cu, classVar, true, constructorIndex);
2091    }
```

```
2092 }
2093
2094 //编译类体
2095 static void compileClassBody(CompileUnit* cu, Variable classVar) {
2096
2097     if (matchToken(cu->curParser, TOKEN_STATIC)) {
2098         if (matchToken(cu->curParser, TOKEN_VAR)) {  //处理静态域 static var id
2099             compileVarDefinition(cu, true);
2100         } else {    //处理静态方法, static methodName
2101             compileMethod(cu, classVar, true);
2102         }
2103
2104     } else if (matchToken(cu->curParser, TOKEN_VAR)) {    //实例域
2105         compileVarDefinition(cu, false);
2106     } else {  //类的方法
2107         compileMethod(cu, classVar, false);
2108     }
2109 }
...略
```

函数 compileMethod 用于编译方法，形参 isStatic 表示所编译的方法是否为静态方法。类中的方法有 3 种（其中 xxx 表示方法名）。

（1）method: 如 xxx(arg,...)

（2）getter: 如 xxx 和[index]

（3）setter: 如 xxx=(value)和[index]=(value)

另外，运算符方法也算是 getter，如+(arg)。

进入函数时先置 cu->enclosingClassBK->inStatic 为 isStatic，此时 cu 是模块的编译单元。接下来用指针 methodSign 指向 token 的签名函数，一会用它创建方法的签名。第 2035 ～2039 行创建并初始化签名结构，签名函数会用到此结构。方法是独立的指令流，因此需要单独的编译单元。下面创建 methodCU 用于编译方法。将当前编译单元 cu 作为 methodCU 的直接外层以形成作用域嵌套链，注意，传给 initCompileUnit 的第四个参数是 true，因为我们在编译方法，不是函数。第 2051 行调用 methodSign 创建签名。接下来读取方法体的左括号，第 2055 行判断构造函数前面不能加关键字 static。接下来生成签名为字符串，存储到 signatureString 中。第 2063 行调用函数 declareMethod 声明所编译的方法，接下来调用 compileBody 编译方法体。编译方法体后，调用 endCompileUnit 把方法体生成闭包（由其中生成的 OPCODE_CREATE_CLOSURE 完成）放到栈顶，然后第 2077 行调用 defineMethod 把方法绑定到类中。注意，此时的方法是绑定到对象所属类的 methods 中，即当前方法是实例方法，即使是构造函数 new，此时也是实例方法。下面针对构造函数做进一步的处理，即生成之前讨论过的静态方法 new。对于构造函数来说，第 2084 行的 constructorIndex 和 methodIndex 的值是一样的，都是 vm->allMethodNames 中的索引。第 2087 行调用 emitCreateInstance 为构造函数创建静态方法，下面再次调用 defineMethod 把方法绑定到类的类中，即元类的 methods 中，静态方法都放在元类中。

下面的函数 compileClassBody 用于编译类体，类体如下形式。

```
class Foo{
    var instantField
    static var staticField
    instantField(){
```

```
    }
    static staticMethod(){
    }
    new(){
    }
}
```

用关键字 static 修饰的都是静态域或静态方法，compileClassBody 也是按照此形式做各种匹配调用。

6.20.4　编译类定义

本节编译类定义，会调用之前介绍的函数。

stepByStep/c6/r/compiler/compiler.c

```
...略
2111 //编译类定义
2112 static void compileClassDefinition(CompileUnit* cu) {
2113     Variable classVar;
2114     if (cu->scopeDepth != -1) { //目前只支持在模块作用域定义类
2115         COMPILE_ERROR(cu->curParser,
2116             "class definition must be in the module scope!");
2117     }
2118
2119     classVar.scopeType = VAR_SCOPE_MODULE;
2120     consumeCurToken(cu->curParser, TOKEN_ID,
2121         "keyword class should follow by class name!");  //读入类名
2122     classVar.index = declareVariable(cu,
2123             cu->curParser->preToken.start, cu->curParser->preToken.length);
2124
2125     //生成类名,用于创建类
2126     ObjString* className = newObjString(cu->curParser->vm,
2127             cu->curParser->preToken.start, cu->curParser->preToken.length);
2128
2129     //生成加载类名的指令
2130     emitLoadConstant(cu, OBJ_TO_VALUE(className));
2131     if (matchToken(cu->curParser, TOKEN_LESS)) {    //类继承
2132         expression(cu, BP_CALL);    //把父类名加载到栈顶
2133     } else {    //默认加载 object 类为基类
2134         emitLoadModuleVar(cu, "object");
2135     }
2136
2137     //创建类需要知道域的个数,目前类未定义完,因此域的个数未知,
2138     //因此先临时写为 255,待类编译完成后再回填属性数
2139         int fieldNumIndex = writeOpCodeByteOperand(cu, OPCODE_CREATE_CLASS,
255);
2140
2141     //虚拟机执行完 OPCODE_CREATE_CLASS 后,栈顶留下了创建好的类,
2142     //因此现在可以用该类为之前声明的类名 className 赋值
2143     if (cu->scopeDepth == -1) {
```

```
2144            emitStoreModuleVar(cu, classVar.index);
2145        }
2146
2147        ClassBookKeep classBK;
2148        classBK.name = className;
2149        classBK.inStatic = false;    //默认为 false
2150        StringBufferInit(&classBK.fields);
2151        IntBufferInit(&classBK.instantMethods);
2152        IntBufferInit(&classBK.staticMethods);
2153
2154        //此时 cu 是模块的编译单元,跟踪当前编译的类
2155        cu->enclosingClassBK = &classBK;
2156
2157        //读入类名后的{
2158        consumeCurToken(cu->curParser, TOKEN_LEFT_BRACE,
2159            "expect '{' after class name in the class declaration!");
2160
2161    //进入类体
2162        enterScope(cu);
2163
2164        //直到类定义结束}为止
2165        while (!matchToken(cu->curParser, TOKEN_RIGHT_BRACE)) {
2166            compileClassBody(cu, classVar);
2167            if (PEEK_TOKEN(cu->curParser) == TOKEN_EOF) {
2168                COMPILE_ERROR(cu->curParser, "expect '}' at the end of class declaration!");
2169            }
2170        }
2171
2172        //上面临时写了 255 个字段,现在类编译完成,回填正确的字段数
2173        //classBK.fields 的是由 compileVarDefinition 函数统计的
2174        cu->fn->instrStream.datas[fieldNumIndex] = classBK.fields.count;
2175
2176        symbolTableClear(cu->curParser->vm, &classBK.fields);
2177        IntBufferClear(cu->curParser->vm, &classBK.instantMethods);
2178        IntBufferClear(cu->curParser->vm, &classBK.staticMethods);
2179
2180        //enclosingClassBK 用来表示是否在编译类,
2181        //编译完类后要置空,编译下一个类时再重新赋值
2182        cu->enclosingClassBK = NULL;
2183
2184        //退出作用域,丢弃相关局部变量
2185        leaveScope(cu);
2186    }
...略
```

函数 compileClassDefinition 用于编译类定义。目前只支持在模块作用域内定义类，所以在开头先进行判断。生成的 classVar 的作用域类型 scopeType 被置为 VAR_SCOPE_MODULE，表示类处于模块作用域。接下来调用 consumeCurToken 读入类名，然后调用 declareVariable 声明类名为模块变量，索引存储到 classVar.index。第 2126 行的 className 是类名的 ObjString 形式，后面会

用到。下面调用 emitLoadConstant 生成加载类名的指令，类名是做常量存储在 cu->fn->constants 中的，此函数会确保类名先被添加到 cu->fn->constants 中。默认情况下 object 类是所有类的基类，如果类有指定的基类，那么需要以符号<指定，形式如 class subClass < baseClass。第 2131 行判断若后面有符号<，说明有显式的基类，于是调用 expression 加载基类，否则第 2134 行加载 object 为基类。object 类是所有类的基类，我们在 core.c 中的函数 buildCore 中创建了它。执行到 2139 行时，栈顶是基类，次栈顶是类名，再指定实例域个数就可以创建类了，创建类的操作码是 OPCODE_CREATE_CLASS（这就是之前所说的“编译器处理构造函数过程”的第一步），但是目前仍处于类编译中，只有在编译完了类定义之后才会知道实例域个数，因此先临时写为 255，待类编译完成后再回填。虚拟机执行完 OPCODE_CREATE_CLASS 后，会在栈顶留下创建好的类，因此现在可以用该类为之前声明的类名 className 赋值了，暂不支持嵌套类定义，类只能是模块变量。第 2143 行判断如果当前是模块作用域，就把栈顶的类同步回类名所在的模块变量中，作为模块变量的值。说明一下，局部变量的值是存储在运行时栈，因此无需同步。

第 2147～2152 行是定义并初始化 ClassBookKeep 结构 classBK，它用来在编译类时跟踪类信息，并在初始化之后使 cu->enclosingClassBK 指向它。第 2162 行调用 enterScope 使作用域增加，表示进入类体进行方法和域的编译。以上所做的工作是和类名有关的部分，第 2165 行的 while 循环用于编译类体，调用前面的 compileClassBody 函数进行编译。编译过后，知道了实例域的个数，于是在第 2174 行回填域个数。注意，回填只是个概念，并不是非要用诸如 patchxxx 之类的函数完成，知道往哪写入正确的值即可。再后面就是一些清理工作。

6.21 编译函数定义（本节源码 stepByStep/c6/s）

本语言完全面向对象，按照面向对象的方式来定义函数，形式如：

```
var  functionName = Fn.new {|形参列表|
     函数体代码
}
```

此函数调用形式 functionName.call（实参列表）。

如果按照传统的函数定义，我们的函数定义的形式如：

```
fun  functionName (形参列表) {
     函数体代码
}
```

此函数调用形式如 Name（实参列表）。

我们两种定义形式都提供，第二种的关键字 fun 表示 function，function 太长了，基于简单的原则我们用 fun 来代替 function。本节完成用关键字 fun 定义函数。

对于第二种来说仅仅是语法糖，在编译上我们按照第一种形式编译就行了。但是通过 fun 关键字定义的函数只能在类外定义，如果在类内定义，就与类中的方法调用和方法定义混淆了，而且也在方法中定义函数感觉很奇怪，毕竟可以用定义方法来代替，因此我们在语法上规定：对于用关键字 fun 定义的函数，只能在类外定义并且只在类外调用，确切来说是在模块作用域中定义与调用。这么做是有原因的，比如对于某方法中的 aaa()，我们（具体是 id 函数）分不清这是调用本类方法 aaa 还是调用函数 aaa，尽管在编译结束后可以先按照方法来查找，再按照函数来查找，但这严重降低了效率。如果非要在方法中定义并调用函数，可以用第一种 Fn.new 的方式创建，然后用.call 的方式调用。

stepByStep/c6/s/compiler/compiler.c

```
...略
//编译函数定义
static void compileFunctionDefinition(CompileUnit* cu) {
    // 本语言完全面向对象,
    // (一)
    //      函数定义的形式是:
    //          var function = Fn.new {|形参|
    //              函数体代码
    //          }
    //      Fn.new 返回的是{}内函数的闭包

    //      函数调用的形式是 "函数闭包.call(...)"

    //      但是传统上
    // (二)
    //      函数定义的形式是:
    //          function 函数名(形参) {
    //              函数体代码
    //          }
    //      函数调用形式是:
    //          函数名(实参)

    //fun 关键字只用在模块作用域中
    if (cu->enclosingUnit != NULL) {
        COMPILE_ERROR(cu->curParser, "'fun' should be in module scope!");
    }

        consumeCurToken(cu->curParser, TOKEN_ID, "missing function name!");

    //函数名加上 Fn 前缀作为模块变量存储
    char fnName[MAX_SIGN_LEN + 4] = {'\0'}; //"fn xxx\0"
    memmove(fnName, "Fn ", 3);  //把函数名加上 Fn 前缀后作为变量名
    memmove(fnName + 3, cu->curParser->preToken.start,
        cu->curParser->preToken.length);

    uint32_t fnNameIndex = declareVariable(cu, fnName, strlen(fnName));

    //生成 fnCU,专用于存储函数指令流
    CompileUnit fnCU;
    initCompileUnit(cu->curParser, &fnCU, cu, false);

    Signature tmpFnSign = {SIGN_METHOD, "", 0, 0}; //临时用于编译函数
    consumeCurToken(cu->curParser,
            TOKEN_LEFT_PAREN, "expect '(' after function name!");

    //若有形参则将形参声明局部变量
    if (!matchToken(cu->curParser, TOKEN_RIGHT_PAREN)) {
        processParaList(&fnCU, &tmpFnSign);       //将形参声明为函数的局部变量
        consumeCurToken(cu->curParser,
```

```
2236                TOKEN_RIGHT_PAREN, "expect ')' after parameter list!");
2237        }
2238
2239        fnCU.fn->argNum = tmpFnSign.argNum;
2240
2241        consumeCurToken(cu->curParser,
2242              TOKEN_LEFT_BRACE, "expect '{' at the beginning of method body.");
2243
2244        //编译函数体,将指令流写进该函数自己的指令单元 fnCu
2245        compileBody(&fnCU, false);
2246
2247 #if DEBUG
2248        endCompileUnit(&fnCU, fnName, strlen(fnName));
2249 #else
2250        endCompileUnit(&fnCU);
2251 #endif
2252
2253        //将栈顶的闭包写入变量
2254        defineVariable(cu, fnNameIndex);
2255 }
…略
```

函数 compileFunctionDefinition 用于编译 fun 关键字形式的函数定义。第 2210 行判断 fun 只能出现在模块作用域中。接下来读取函数名。用关键字 fun 定义的函数以模块变量的形式存储到 module->moduleVarName 和 module->moduleVarValue 中，为避免用户自定义的变量和函数名冲突，我们以“Fn 函数名”为该模块变量的名字存储该函数，以后使用函数时要人为加上前缀 Fn，我们前面介绍的函数 id 中就是这样做的。第 2217～2220 行为函数名加上前缀 Fn 后存储到 fnName 中，接着调用 declareVariable 将函数名 fnName 作为模块变量声明。接下来生成 fnCU，专门用来存储函数的指令流。第 2228 行生成临时的签名结构 tmpFnSign 用于统计形参个数。第 2233 行判断若有形参列表，就调用 processParaList 处理，形参个数录入到 tmpFnSign 中。第 2239 行使 fnCU.fn->argNum 等于 tmpFnSign.argNum，然后读取代码块开头的标识“{”。第 2245 行调用 compileBody 编译函数体。

编译后调用 endCompileUnit 将函数体生成闭包，最后调用 defineVariable 函数将栈顶的闭包（由 endCompileUnit 生成的由 CREATE_CLOSURE 指令创建并置于栈顶）同步到函数所在的模块变量中。

6.22 编译模块导入（本节源码 stepByStep/c6/t）

模块化是程序有效的组织方式，当需要一个模块时可以导入，导入模块的关键字是 import，导入模块就是执行这个模块，文件就是模块。

```
[work@work spr]$ cat -n /tmp/test.sp
     1var a = 9
     2System.print("from module test.")
[work@work spr]$ cat -n /tmp/t.sp
     1import test
[work@work spr]$ ./spr /tmp/t.sp
from module test.
[work@work spr]$
```

不过此种导入形式是无法使用模块 test 中的模块变量 a，比如以下引用变量 a 时会报错。

```
[work@work spr]$ cat -n /tmp/test.sp
     1var a = 9
 [work@work spr]$ cat -n /tmp/t.sp
     1import test
     2System.print(a)
[work@work spr]$ ./spr /tmp/t.sp
/tmp/t.sp: "line:2, variable 'a' not defined!"
[work@work spr]$
```

如果想导入模块中的模块变量，用 for 关键字单独指定，for 后面接的是以逗号分隔的模块变量列表（传统语言中列表中的元素都以逗号分隔）。值得注意的是，无论是否导入模块，只要 import 模块，该模块都会被执行（其他语言也是如此）。比如以下示例。

```
[work@work spr]$ cat -n /tmp/test.sp
     1var a = 9
     2System.print("from module test.")
[work@work spr]$ cat -n /tmp/t.sp
     1import test for a
     2fun tt() {
     3    System.print(a)
     4}
     5
     6tt()
[work@work spr]$ ./spr /tmp/t.sp
from module test.
9
[work@work spr]$
```

以上示例中导入的是模块 test 中的模块变量 a。被导入的都是模块中的模块变量，不可能导入模块中的局部变量。不过模块变量被导入后是模块变量还是局部变量，这取决于 import 关键字所在的作用域，即想把模块中的模块变量作为何种类型变量导入，就把关键字 import 放在哪种作用域中。比如下面的例子就是把模块 test 中的模块变量 a 导入到函数 tt 中，因此 a 就作为函数 tt 中的局部变量。

```
[work@work spr]$ cat -n /tmp/test.sp
     1var a = 9
     2System.print("from module test.")
[work@work spr]$ cat -n /tmp/t.sp
     1fun tt() {
     2    import test for a
     3    System.print(a)
     4}
     5
     6tt()
[work@work spr]$ ./spr /tmp/t.sp
from module test.
9
[work@work spr]$
```

强调，被导入的模块变量将变成 import 所在作用域中的变量类型，为验证这一点，大家看下面的例子。

```
[work@work spr]$ cat -n /tmp/test.sp
     1var a = 9
     2System.print("from module test.")
[work@work spr]$ cat -n /tmp/t.sp
     1fun tt() {
     2    import test for a
     3    System.print(a)
     4}
     5
     6tt()
     7System.print(a)
[work@work spr]$ ./spr /tmp/t.sp
/tmp/t.sp:"line:7, variable 'a' not defined!"
[work@work spr]$
```

你看，由于模块变量 a 只导入到了函数 tt 中，第 7 行访问不到 a，编译器就报错了。也许有读者会问为什么不再输出 from module test.，原因是我们并不是像 shell 脚本或批处理脚本那样“边解释边执行”，我们是在编译之后才执行，报错是在编译阶段，报错后就退出编译了，而输出 from module test.是在执行阶段，虚拟机尚未来得及执行就被前面的报错结束了。

对于上面的情况，可以在最外层再加一句 import test for a 就没问题了，比如下面。

```
[work@work spr]$ cat -n /tmp/test.sp
     1var a = 9
     2System.print("from module test.")
[work@work spr]$ cat -n /tmp/t.sp
     1import test for a
     2fun tt() {
     3    import test for a
     4    a = 7
     5    System.print(a)
     6}
     7
     8tt()
     9System.print(a)
[work@work spr]$ ./spr /tmp/t.sp
from module test.
7
9
[work@work spr]$
```

在函数 tt 内部特意将导入的变量 a 重新赋值为 7，这和外部导入的不冲突，因此输出 7 和 9。

例子举得够多了，下面看真正的代码。

stepByStep/c6/t/compiler/compiler.c

```
...略
2257 //编译 import 导入
2258 static void compileImport(CompileUnit* cu) {
2259 //     import "foo"
```

```
2260 //将按照以下形式处理:
2261 //  System.importModule("foo")
2262
2263 //   import foo for bar1, bar2
2264 //将按照以下形式处理:
2265 //  var bar1 = System.getModuleVariable("foo", "bar1")
2266 //  var bar2 = System.getModuleVariable("foo", "bar2")
2267
2268      consumeCurToken(cu->curParser, TOKEN_ID, "expect module name after export!");
2269
2270      //备份模块名 token
2271      Token moduleNameToken = cu->curParser->preToken;
2272
2273      //导入时模块的扩展名不需要,有可能用户会把模块的扩展名加上,
2274      //比如 import sparrow.sp, 这时候就要跳过扩展名
2275      if (cu->curParser->preToken.start[cu->curParser->preToken.length] == '.') {
2276           printf("\nwarning!!! the imported module needn't extension!,
             compiler try to ignor it!\n");
2277
2278         //跳过扩展名
2279         getNextToken(cu->curParser); //跳过.
2280         getNextToken(cu->curParser); //跳过 sp
2281      }
2282
2283      //把模块名转为字符串,存储为常量
2284      ObjString* moduleName = newObjString(cu->curParser->vm,
2285           moduleNameToken.start, moduleNameToken.length);
2286                   uint32_t constModIdx = addConstant(cu, OBJ_TO_VALUE(moduleName));
2287
2288      //1. 为调用 system.importModule("foo")压入参数 system
2289      emitLoadModuleVar(cu, "System");
2290      //2. 为调用 system.importModule("foo")压入参数 foo
2291               writeOpCodeShortOperand(cu, OPCODE_LOAD_CONSTANT, constModIdx);
2292      //3. 现在可以调用 system.importModule("foo")
2293      emitCall(cu, 1, "importModule(_)", 15);
2294
2295      //回收返回值 args[0]所在空间
2296      writeOpCode(cu, OPCODE_POP);
2297
2298      //如果后面没有关键字 for 就导入结束
2299      if (!matchToken(cu->curParser, TOKEN_FOR)) {
2300           return;
2301      }
2302
2303      //循环编译导入的模块变量(以逗号分隔),
2304      //如 import foo for bar1, bar2 中的 bar1, bar2
2305      do {
2306         consumeCurToken(cu->curParser, TOKEN_ID,
2307              "expect variable name after 'for' in import!");
2308
```

```
2309          //在本模块中声明导入的模块变量
2310          uint32_t varIdx = declareVariable(cu,
2311                                            cu->curParser->preToken.start, cu->
curParser->preToken.length);
2312
2313          //把模块变量转为字符串,存储为常量
2314          ObjString* constVarName = newObjString(cu->curParser->vm,
2315                                                 cu->curParser->preToken.start, cu->
                                curParser->preToken.length);
2316                               uint32_t constVarIdx = addConstant(cu, OBJ_TO_VALUE
                                (constVarName));
2317
2318          // 1. 为调用 System.getModuleVariable("foo", "bar1") 压入 system
2319          emitLoadModuleVar(cu, "System");
2320          // 2. 为调用 System.getModuleVariable("foo", "bar1") 压入 foo
2321                writeOpCodeShortOperand(cu, OPCODE_LOAD_CONSTANT, constModIdx);
2322          // 3. 为调用 System.getModuleVariable("foo", "bar1") 压入 bar1
2323                writeOpCodeShortOperand(cu, OPCODE_LOAD_CONSTANT, constVarIdx);
2324          // 4. 调用 System.getModuleVariable("foo", "bar1")
2325          emitCall(cu, 2, "getModuleVariable(_,_)", 22);
2326
2327          //此时栈顶是 system.getModuleVariable("foo", "bar1")的返回值,
2328          //即导入的模块变量的值，下面将其同步到相应变量中
2329          defineVariable(cu, varIdx);
2330      } while (matchToken(cu->curParser, TOKEN_COMMA));
2331 }
…略
```

函数 compileImport 用于编译模块导入。我们已经知道，模块导入的语法是 import foo for bar1, bar2，for 后面的是待导入的、以逗号分隔的模块变量列表，可以省略，省略后就表示只导入模块，导入模块就是执行这个模块。但其实这也是语法糖，我们在“幕后”会按照以下形式编译。

```
import "foo"
```

将按照以下形式处理：

```
System.importModule("foo")
```

而

```
import foo for bar1, bar2
```

将按照以下形式处理：

```
var bar1 = System.getModuleVariable("foo", "bar1")
var bar2 = System.getModuleVariable("foo", "bar2")
```

System 类是我们在 core.script.inc 中实现的类，此文件是脚本代码，并且预定义了一些方法，不过那是用脚本语言写的方法，而这里所说的方法 importModule 和 getModuleVariable 是我们在 buildCore 函数中为 System 类添加的原生方法，即是用 C 实现的，其中 importModule 的作用就是执行被导入的模块，很快我们就会完善 buildCore 函数，到时候大家就清楚了。另外，我们并不要求用户输入的模块名和模块变量是字符串，因此 import 和 for 后面的关键字并不需要加双引号，

不过我们会将其转化为字符串处理。

第 2268 行读入模块名，读入之后 preToken 便是模块名。接下来用 moduleNameToken 备份模块名，这是担心用户在导入模块时添加了扩展名.sp，比如上面例子中的导入可以改为 import test.sp for a。正常情况下被导入的模块不需要添加扩展名，若是导入的模块加了扩展名，我们也要容错，这就需要跳过扩展名的 token，那时 preToken 便是“.”，因此不能用 preToken 代替模块名，故在此备份模块名。第 2275～2281 行便是跳过扩展名。接下来把模块名转为字符串，存储后为常量。第 2289～2293 行是按照前面所说的“幕后”形式编译，为调用函数 System.importModule("foo") 生成 3 条指令，第一条指令的目的是压入类 System，第二条指令的目的是压入形参 foo，最后生成调用 System. importModule("foo")的指令，也就是前两条指令是为第 3 条的函数调用在栈中准备参数。

并不是所有的函数都有返回值，但为了方便处理和栈统一，默认都有返回值，且存储到 args[0]（位于栈顶）。无论被调函数是否有返回值，args[0]都会保留下来，若有返回值就存储到 args[0]，否则就空着 args[0]，因此如果不知道调用者是否需要此返回值，那就由调用者来回收位于栈顶的 args[0]的空间——即栈中的 args[0]所占的 slot。前面我们说过了，主调方和被调方的运行时栈是接壤的，被调方的栈底就是主调方的栈顶，因此主调方可以获取被调方的返回值。第 2296 行生成的 OPCODE_POP 就是为了回收位于栈顶的返回值的空间。

如果后面没有关键字 for，就表示导入整个模块，也就是执行整个模块，注意不是导入所有模块变量，此时可以返回了。如果后面有关键字 for 就是导入模块变量，但模块依然也会执行，这是确定的，因为前面已经生成了执行 importModule 方法的代码，该方法就是执行被导入的模块。

被导入的模块变量是以逗号分隔的，后面的 do-while 循环以逗号分隔循环处理模块变量。在第 2306 行读入模块变量名，之后声明此变量。第 2314～2316 行将模块变量名封装为字符串并存储为常量，下面按第二种形式的“幕后”处理编译，最终生成调用 System.getModuleVariable("foo", "bar")的指令。在虚拟机执行过此调用指令之后，栈顶是 system.getModuleVariable("foo", "bar")的返回值，即导入的变量，调用 defineVariable 函数将其存储到上面定义的变量（全局或局部变量）。

语义分析就是用来生成指令，因此语义分析的前提是理解指令，了解了指令的行为才能知道怎样利用指令组合出高级语言的意图。我们之前只是介绍了指令（操作码及其操作数），如果没有对其实现有所了解的话，我们对本章的语义分析就理解不透。因此对于大多数读者来讲，本章的内容可能有种“不懂，但是又说不出来哪里不懂”的感觉，因此我们先去实现虚拟机，让大家先看看虚拟机对各种指令的处理，大家就会明白语义分析——哦，原来是这个样子。

以上有关编译的部分完成了，还有一部分原生方法要补充，先到此为止，将来完成虚拟机后再继续。

第7章 虚拟机

虚拟机没那么复杂，无论是模拟体系架构的虚拟机还是本章中用于执行自定义指令的虚拟机，都是循环地读取指令流，即根据读取到的操作码读取相应的操作数，按照语义执行。

7.1 创建类与堆栈框架（本节源码 stepByStep/c7/a）

类即是 class，对于脚本中定义的类，我们在幕后就要为其创建一个真实的类。再有，方法和函数的运行都离不开运行时栈，本小节先完成这两个任务。

stepByStep/c7/a/object/class.c

```
...略
65 //创建一个类
66 Class* newClass(VM* vm, ObjString* className, uint32_t fieldNum, Class* superClass) {
67     //10 表示 strlen(" metaClass"
68     #define MAX_METACLASS_LEN MAX_ID_LEN + 10
69     char newClassName[MAX_METACLASS_LEN] = {'\0'};
70     #undef MAX_METACLASS_LEN
71
72     memcpy(new-ClassName, className->value.start, className->value.length);
73     memcpy(newClassName + className->value.length, " metaclass", 10);
74
75     //先创建子类的 meta 类
76     Class* metaclass = newRawClass(vm, newClassName, 0);
77     metaclass->objHeader.class = vm->classOfClass;
78
79     //绑定 classOfClass 为 meta 类的基类
80     //所有类的 meta 类的基类都是 classOfClass
81     bindSuperClass(vm, metaclass, vm->classOfClass);
82
83     //最后再创建类
84     memcpy(newClassName, className->value.start, className->value.length);
85     newClassName[className->value.length] = '\0';
86     Class* class = newRawClass(vm, newClassName, fieldNum);
87
88     class->objHeader.class = metaclass;
89     bindSuperClass(vm, class, superClass);
90
91     return class;
92 }
...略
```



函数 newClass 就是用于创建一个类，类名为 className，域个数为 fieldNum，基类为 superClass。在函数前部是为类的 meta 类准备名字，即在类名后追加 metaClass。接下来是一些创建类的准备工作，先调用 newRawClass 创建类的 meta 类，然后第 77 行指定 meta 类的 meta 类为 classOfClass，随后调用 bindSuperClass 使 meta 类的基类为 classOfClass。有了 meta 类后，最后才是创建类自身，调用 newRawClass 创建类，指定其 meta 类为上面为其准备的 metaclass，绑定为 superClass 的子类后返回，自此类创建完成。调用的函数都是之前介绍过的，不再赘述。

下面是实现运行时栈部分。

stepByStep/c7/a/vm/vm.c

```
...略
25 //确保 stack 有效
26 void ensureStack(VM* vm, ObjThread* objThread, uint32_t neededSlots) {
27     if (objThread->stackCapacity >= neededSlots) {
28         return;
29     }
30
31     uint32_t newStackCapacity = ceilToPowerOf2(neededSlots);
32               ASSERT(newStackCapacity > objThread->stackCapacity,
                 "newStackCapacity error!");
33
34     //记录原栈底以用于下面判断扩容后的栈是否是原地扩容
35     Value* oldStackBottom = objThread->stack;
36
37     uint32_t slotSize = sizeof(Value);
38     objThread->stack = (Value*)memManager(vm, objThread->stack,
39               objThread->stackCapacity * slotSize, newStackCapacity * slotSize);
40     objThread->stackCapacity = newStackCapacity;
41
42     //为判断是否原地扩容
43     long offset = objThread->stack - oldStackBottom;
44
45     //说明 os 无法在原地满足内存需求，重新分配了起始地址，下面要调整
46     if (offset != 0) {
47         //调整各堆栈框架的地址
48         uint32_t idx = 0;
49         while (idx < objThread->usedFrameNum) {
50             objThread->frames[idx++].stackStart += offset;
51         }
52
53         //调整 open upValue
54         ObjUpvalue* upvalue = objThread->openUpvalues;
55         while (upvalue != NULL) {
56             upvalue->localVarPtr += offset;
57             upvalue = upvalue->next;
58         }
59
60         //更新栈顶
61         objThread->esp += offset;
62     }
```

```
63 }
64
65 //为 objClosure 在 objThread 中创建运行时栈
66 inline static void createFrame(VM* vm, ObjThread* objThread,
67         ObjClosure* objClosure, int argNum) {
68
69     if (objThread->usedFrameNum + 1 > objThread->frameCapacity) { //扩容
70         uint32_t newCapacity = objThread->frameCapacity * 2;
71         uint32_t frameSize = sizeof(Frame);
72                         objThread->frames = (Frame*)memManager(vm, objThread->frames,
73                         frameSize * objThread->frameCapacity, frameSize * newCapacity);
74         objThread->frameCapacity = newCapacity;
75     }
76
77     //栈大小等于栈顶-栈底
78     uint32_t stackSlots = (uint32_t)(objThread->esp - objThread->stack);
79     //总共需要的栈大小
80     uint32_t neededSlots = stackSlots + objClosure->fn->maxStackSlotUsedNum;
81
82     ensureStack(vm, objThread, neededSlots);
83
84     //准备上 CPU
85     prepareFrame(objThread, objClosure, objThread->esp - argNum);
86 }
...略
```

函数 ensureStack 从名字上是“确保栈有效”，这里确保的是栈的容量及数据有效，其中形参 neededSlots 是指栈最少具有的容量。因此函数开头便判断栈容量 stackCapacity 是否大于所要求的 neededSlots，如果是就返回，无需调整。第 31 行的 newStackCapacity 是栈的新容量。

第 43 行的 offset 是为判断是否需要原地扩容。内存管理系统负责把虚拟地址与物理地址做映射，虚拟地址所映射的物理地址可以不连续，但虚拟地址是逻辑上的概念，其肯定是连续的。申请的内存返回的是虚拟地址，用户程序使用的就是这种连续的虚拟地址，内存管理系统为满足 realloc 这种扩容分配（当然也可用于减少内存空间），往往在所分配的空间上预留了一部分空间以满足将来的原地扩容，这就导致了内存池被分配得七零八落，那些预留的空间就成了“碎片”。虚拟地址是连续的，如果在扩容时增量的大小还在预留的空间内，就保证了虚拟地址的连续性。当增量超过了预留空间的大小，如果依然想利用该预留空间，则必然要再寻一块空间以补足差缺，但这破坏了虚拟地址的连续性，故此方案不成立。一个好的内存管理系统应该尽可能地让使用者感到方便，用户需要的是一片大小为 xxx 且连续（虚拟地址）的内存块。因此，当预留的剩余空间无法满足原地扩容时的增量时，就不利用该剩余空间了，而是重新找一块更大且连续的空间以满足扩容需求。当内存管理系统找到合适的内存空间时，就会把原来空间中的数据拷贝过去并返回新分配的内存起始地址，至此内存管理系统的工作已经完成了。你看，内存大小也满足了，原来内存块中的数据也拷贝过去了。但是，如果涉及指向原内存块的指针，由于内存块已经被替换成了新的内存块，也就是说地址变了，相关指针就要更新为新的地址，这样才不会出错。因此，如果涉及原内存块中的指针，用户进程肯定要调整该指针的值以使其指向正确的位置。总结一下，内存池分配尽量在原地扩展，但如果上面的剩余空间不足以容纳内存增量的话，内存管理系统便会重新找一块足够广阔的内存空间以满足内存需求，并同时把原内存空间中的数据拷贝过去，切



记，这是内存管理系统自动完成的，不需要用户干涉。

新分配的内存块起始地址肯定不与原来的相同，因此第 46 行判断，如果 offset 不为 0，说明操作系统（内存管理系统）无法满足原地扩容，已经重新分配了内存，因此我们要调整相关指针了。涉及内存地址的指针有 3 方面：线程中各堆栈框架的起始地址 stackStart，upvalue 中的 localVarPtr 指向的内存块中的内存地址（栈中 slot 地址），以及线程的栈顶 esp 也要更新。下面就是分别在这 3 方面做了修补，即加上 offset 即可。

第 66 行的函数 createFrame 用于为 objClosure 在 objThread 中运行而创建堆栈框架，堆栈构架即是闭包（函数或方法）的运行资源环境。堆栈框架中包括哪些内容呢？包括待运行的指令流（函数或方法），和当前运行的指令地址及运行时栈。前两个不用多说了，说说运行时栈。运行时栈就是一个栈，是内存中的一块区域，我们是用 Value 数组来模拟它的。运行时栈起始地址是记录在堆栈框架 frame 中。frame 是记录在线程中，一个线程就是一个任务调度单位，它是独立于其他任务的。想想看，线程是什么？就是一个单独的执行流，而执行流就是可被任务调度器单独调度的指令流，比如一个进程也是单独的执行流。指令流其实就是函数或方法，因为我们在创建线程时都是以一个函数为参数，就是说让线程运行这个函数。因此简单来说，线程就是函数的容器。方法或函数运行时肯定需要运行时栈，函数都有自己的指令流，进而需要堆栈框架去跟踪这些，因此线程要负责分配堆栈框架 frame。线程中运行的函数还可能会调用其他函数，每个函数都要有单独的运行时栈，因此在线程中需要管理多个 frame，这就是线程中有 frame 数组 frames 的原因。frame 的数量是有限的，数量记录在 objThread->usedFrameNum 中。第 69 行判断 frame 是否够用，即是否需要扩容，第 70～74 行是相应的扩容过程。本函数是为调用闭包（形参 objClosure）而创建相应的运行环境等资源，该闭包运行时必然需要占用一定量的栈空间，比如局部变量和指令对栈的影响，我们在编译过程中就把闭包所需要的栈大小统计到变量 objClosure->fn->maxStackSlotUsedNum 中了。复习一下，线程提供了一个“大栈”，在线程中运行的函数们共享这一块大栈，各自使用其中的一部分作为运行时栈。接下来计算栈的容量，第 78 行获得线程中“大栈”的容量尺寸，再加上所运行闭包所需要的栈容量，使用这两者之和作为 ensureStack 函数的参数以保证栈容量及数据正确性。最后调用之前介绍过的 prepareFrame（定义在 obj_thread.c 中）使运行时栈及 ip 指针就绪，也就是准备让形参 objClosure 上 CPU 运行。说明一下，prepareFrame 的第 3 个参数是被调用函数的堆栈框架在栈中的起始地址，这里减去了参数个数 argNum，目的是让被调用的闭包函数 objClosure 可以访问到栈中自己的参数。其中 argNum 占用的空间在编译过程中已经算在了 maxStackSlotUsedNum 之内了。

7.2 upvalue 的创建与关闭（本节源码 stepByStep/c7/b）

内层作用域在引用外层作用域中的局部变量时，该局部变量对内层作用域来说就是 upvalue，总之 upvalue 是外层作用域中的局部变量，upvalue 是如何组织的呢？本节介绍 upvalue 的创建与关闭。

stepByStep/c7/b/vm/vm.c (1/2)

```
...略
88 //关闭在栈中 slot 为 lastSlot 及之上的 upvalue
89 static void closedUpvalue(ObjThread* objThread, Value* lastSlot) {
90       ObjUpvalue* upvalue = objThread->openUpvalues;
91       while (upvalue != NULL && upvalue->localVarPtr >= lastSlot) {
92          //localVarPtr 改指向本结构中的 closedUpvalue
93          upvalue->closedUpvalue = *(upvalue->localVarPtr);
94          upvalue->localVarPtr = &(upvalue->closedUpvalue);
```




```
95
96          upvalue = upvalue->next;
97      }
98      objThread->openUpvalues = upvalue;
99  }
100
101 //创建线程已打开的 upvalue 链表，并将 localVarPtr 所属的 upvalue 以降序插入到该链表
102 static ObjUpvalue* createOpenUpvalue(VM* vm, ObjThread* objThread, Value*
    localVarPtr) {
103     //如果 openUpvalues 链表为空就创建
104     if (objThread->openUpvalues == NULL) {
105         objThread->openUpvalues = newObjUpvalue(vm, localVarPtr);
106         return objThread->openUpvalues;
107     }
108
109     //下面以 upvalue.localVarPtr 降序组织 openUpvalues
110     ObjUpvalue* preUpvalue = NULL;
111     ObjUpvalue* upvalue = objThread->openUpvalues;
112
113     //后面的代码保证了 openUpvalues 按照降序组织，
114     //下面向堆栈的底部遍历，直到找到合适的插入位置
115     while (upvalue != NULL && upvalue->localVarPtr > localVarPtr) {
116        preUpvalue = upvalue;
117        upvalue = upvalue->next;
118     }
119
120     //如果之前已经插入了该 upvalue 则返回
121     if (upvalue != NULL && upvalue->localVarPtr == localVarPtr) {
122         return upvalue;
123     }
124
125     //openUpvalues 中未找到该 upvalue，
126     //现在就创建新 upvalue，按照降序插入到链表
127     ObjUpvalue* newUpvalue = newObjUpvalue(vm, localVarPtr);
128
129     //保证了 openUpvalues 首节点 upvalue->localVarPtr 的值是最高的
130     if (preUpvalue == NULL) {
131         //说明上面 while 的循环体未执行，新节点（形参 localVarPtr）的值大于等于链表首节点
132         //因此使链表节点指向它所在的新 upvalue 节点
133         objThread->openUpvalues = newUpvalue;
134     } else {
135         //preUpvalue 已处于正确的位置
136         preUpvalue->next = newUpvalue;
137     }
138     newUpvalue->next = upvalue;
139
140     return newUpvalue;//返回该节点
141 }
```

…略



函数 closedUpvalue 用于关闭在栈中 slot 为 lastSlot 及之上的 upvalue。

open upvalue 是所指向的局部变量还在栈中的 upvalue，不在栈中的 upvalue 称为 closed upvalue。但 objThread->openUpvalues 是指所有在本线程中“已经打开过的 upvalue”的链表首节点，即同时包括 open upvalue 和 closed upvalue，因为当 upvalue->localVarPtr 指向的局部变量出栈后，该局部变量会存储到 upvalue->closedUpvalue，这种 upvalue 就是 closed upvalue，因此 objThread->openUpvalues 被称为“已打开的 upvalue”链表。链表节点是个 ObjUpvalue 结构，ObjUpvalue->localVarPtr 是指针，指向位于栈中的局部变量。我们的栈是数组模拟的，发展方向是往高地址方向（并不和 CPU 硬件中栈的发展方向一致）。本函数关闭的是马上要出作用域的局部变量，因此该作用域及其之内嵌套更深作用域的局部变量都应该被回收，地址位于栈顶 lastSlot 后面（地址更大）的肯定作用域更深的局部变量，因此 upvalue->localVarPtr >= lastSlot 才是需要被关闭的局部变量的条件。之前我们说过关闭 upvalue 的思路：当 upvalue 关闭时（upvalue 所指向的局部变量已从栈中销毁时就称为 upvalue 关闭或 closed upvalue），那么该被销毁的局部变量会放到 closedUpvalue 变量中，然后 localVarPtr 由指向运行时栈中的局部变量改为指向本结构中的 closedUpvalue。

下面就是以上思路具体的实现了，第 90～97 行遍历 objThread->openUpvalues 链表，将被关闭的局部变量放在相应 upvalue 的 closedUpvalue 中，然后使 localVarPtr 指向 closedUpvalue 的地址。这就是之前我们说过的“upvalue 从栈中弹出后的容身之所”。也许有读者会想，第 91 行的 while 循环条件似乎有问题，当链表中 upvalue 节点的 localVarPtr 小于 lastSlot 就退出循环了，万一该节点后面的也符合要求，这不就漏掉了吗？是这样的，此链表节点在创建时就保证了顺序，马上我们会在函数 createOpenUpvalue 中看到，这是按照 upvalue->localVarPtr 降序排列的，也就是说链表首节点的 localVarPtr 值最大，即地址最深，那么循环条件中出现 upvalue->localVarPtr < lastSlot 时，其后节点的 localVarPtr 值将更小，因此合理。

函数 createOpenUpvalue，创建线程“已打开的 upvalue”链表，即 openUpvalues 链表，并将 localVarPtr 所属的 upvalue 以降序插入到该链表。openUpvalues 链表是线程中所有打开的 upvalue 链表，为提高遍历链表的效率，我们需要在建立链表的过程中保证节点的顺序，由于经常要处理的是 upvalue->localVarPtr，因此以 upvalue->localVarPtr 降序排序，即链表首节点 upvalue 中指向的局部变量是整个线程的运行时栈中局部变量的地址最高处，当然此地址不一定是栈顶。

函数开头判断若链表首节点 openUpvalues 为空，就调用 newObjUpvalue 创建新节点并使 openUpvalues 指向新节点，然后返回节点。第 109～123 行查找 localVarPtr 所属的 upvalue 是否已在链表中，如果找到了就直接返回该 upvalue 节点。第 125 行以后，说明该节点未存在于链表中，因此创建节点并插入，最后返回新节点，即链表首节点。

stepByStep/c7/b/vm/vm.c (2/2)

```
142
143 //校验基类合法性
144 static void validateSuperClass(VM* vm, Value classNameValue,
145         uint32_t fieldNum, Value superClassValue) {
146
147     //首先确保 superClass 的类型是 class
148     if (!VALUE_IS_CLASS(superClassValue)) {
149         ObjString* classNameString = VALUE_TO_OBJSTR(classNameValue);
150         RUN_ERROR("class \"%s\" 's superClass is not a valid class!",
151                 classNameString->value.start);
152     }
153
154     Class* superClass = VALUE_TO_CLASS(superClassValue);
```

```
155
156     //基类不允许为内建类
157      if (superClass == vm->stringClass ||
158              superClass == vm->mapClass ||
159              superClass == vm->rangeClass ||
160              superClass == vm->listClass ||
161              superClass == vm->nullClass ||
162              superClass == vm->boolClass ||
163              superClass == vm->numClass ||
164              superClass == vm->fnClass ||
165              superClass == vm->threadClass
166              ) {
167        RUN_ERROR("superClass mustn't be a buildin class!");
168     }
169
170     //子类也要继承基类的域,
171     //故子类自己的域+基类域的数量不可超过MAX_FIELD_NUM
172     if (superClass->fieldNum + fieldNum > MAX_FIELD_NUM) {
173           RUN_ERROR("number of field including super exceed %d!", MAX_FIELD_NUM);
174     }
175 }
…略
```

函数 validateSuperClass 用于校验基类合法性，形参 fieldNum 是子类的实例域数量。本函数较简单，首先函数开头确认了基类的类型必须是 class，接下来第 157～165 行判断基类不允许为内建类（不允许从内建类继承）。最后，由于子类的域也要从基类继承，子类域的个数要加上基类域个数，因此要判断不能超过最大域个数 MAX_FIELD_NUM。

7.3 修正操作数（本节源码 stepByStep/c7/c）

我们讨论的是一遍的编译器，总有一些数据是需要在代码全部处理之后才能统计出来，比如子类的 field 数要加上从基类继承的 field 数。可是在类继承时，基类可能定义在后面，我们在编译子类时尚不知道基类 field 数量，只有编译完基类后子类才能确定自己的 field 数目，即编译阶段一过就可以修正一些操作数了，但下面的函数是在运行时阶段执行的，即动态绑定。

stepByStep/c7/c/vm/vm.c

```
…略
177 //修正部分指令操作数
178 static void patchOperand(Class* class, ObjFn* fn) {
179     int ip = 0;
180     OpCode opCode;
181     while (true) {
182        opCode = (OpCode)fn->instrStream.datas[ip++];
183        switch (opCode) {
184
185           case OPCODE_LOAD_FIELD:
186           case OPCODE_STORE_FIELD:
187           case OPCODE_LOAD_THIS_FIELD:
188           case OPCODE_STORE_THIS_FIELD:
```

```
189             //修正子类的 field 数目,参数是 1 字节
190              fn->instrStream.datas[ip++] += class->superClass->fieldNum;
191              break;
192
193          case OPCODE_SUPER0:
194          case OPCODE_SUPER1:
195          case OPCODE_SUPER2:
196          case OPCODE_SUPER3:
197          case OPCODE_SUPER4:
198          case OPCODE_SUPER5:
199          case OPCODE_SUPER6:
200          case OPCODE_SUPER7:
201          case OPCODE_SUPER8:
202          case OPCODE_SUPER9:
203          case OPCODE_SUPER10:
204          case OPCODE_SUPER11:
205          case OPCODE_SUPER12:
206          case OPCODE_SUPER13:
207          case OPCODE_SUPER14:
208          case OPCODE_SUPER15:
209          case OPCODE_SUPER16: {
210          //指令流 1: 2 字节的 method 索引
211          //指令流 2: 2 字节的基类常量索引
212
213             ip += 2; //跳过 2 字节的 method 索引
214             uint32_t superClassIdx =
215              (fn->instrStream.datas[ip] << 8) | fn->instrStream.datas[ip + 1];
216
217             //回填在函数 emitCallBySignature 中的占位 VT_TO_VALUE(VT_NULL)
218             fn->constants.datas[superClassIdx] = OBJ_TO_VALUE(class->superClass);
219
220             ip += 2; //跳过 2 字节的基类索引
221
222             break;
223          }
224
225          case OPCODE_CREATE_CLOSURE: {
226             //指令流: 2 字节待创建闭包的函数在常量表中的索引+函数所用的 upvalue 数 * 2
227
228             //函数是存储到常量表中,获取待创建闭包的函数在常量表中的索引
229             uint32_t fnIdx = (fn->instrStream.datas[ip] << 8) | fn->instrStream.
               datas[ip + 1];
230
231             //递归进入该函数的指令流,继续为其中的 super 和 field 修正操作数
232             patchOperand(class, VALUE_TO_OBJFN(fn->constants.datas[fnIdx]));
233
234             //ip-1 是操作码 OPCODE_CREATE_CLOSURE,
235             //闭包中的参数涉及 upvalue,调用 getBytesOfOperands 获得参数字节数
236             ip += getBytesOfOperands(fn->instrStream.datas, fn->constants.datas, ip - 1);
237
238             break;
```

```
239             }
240
241             case OPCODE_END:
242                 //用于从当前及递归嵌套闭包时返回
243                 return;
244
245             default:
246                 //其他指令不需要回填因此就跳过
247                 ip += getBytesOfOperands(fn->instrStream.datas, fn->constants.datas,
                    ip - 1);
248                 break;
249         }
250     }
251 }
252
253 //绑定方法和修正操作数
254 static void bindMethodAndPatch(VM* vm, OpCode opCode,
255         uint32_t methodIndex, Class* class, Value methodValue) {
256
257     //如果是静态方法,就将类指向 meta 类(使接收者为 meta 类)
258     if (opCode == OPCODE_STATIC_METHOD) {
259         class = class->objHeader.class;
260     }
261
262     Method method;
263     method.type = MT_SCRIPT;
264     method.obj = VALUE_TO_OBJCLOSURE(methodValue);
265
266     //修正操作数
267     patchOperand(class, method.obj->fn);
268
269     //修正过后,绑定 method 到 class
270     bindMethod(vm, class, methodIndex, method);
271 }
```

函数 patchOperand 用于修正一些指令的操作数，注释已经很清楚了。

函数 bindMethodAndPatch 用于绑定方法和修正操作数，也很简单。

7.4 执行指令（本节源码 stepByStep/c7/d）

虚拟机是模拟 CPU 的行为，周而复始地执行指令，因此“执行指令”才是最关键的，本节开始介绍此部分。

7.4.1 一些基础工作

虚拟机部分的代码较长，拆成了几部分，本节介绍一些基础代码，比较简单，方便后面虚拟机操作。

stepByStep/c7/d/vm/vm.c (1/8)

```
...略
273 //执行指令
```



```
274 VMResult executeInstruction(VM* vm, register ObjThread* curThread) {
275     vm->curThread = curThread;
276     register Frame* curFrame;
277     register Value* stackStart;
278     register uint8_t* ip;
279     register ObjFn* fn;
280     OpCode opCode;
281
282     //定义操作运行时栈的宏
283     //esp 是栈中下一个可写入数据的 slot
284     #define PUSH(value)  (*curThread->esp++ = value)    //压栈
285     #define POP()        (*(--curThread->esp))    //出栈
286     #define DROP()       (curThread->esp--)
287     #define PEEK()       (*(curThread->esp - 1)) // 获得栈顶的数据
288     #define PEEK2()      (*(curThread->esp - 2)) // 获得次栈顶的数据
289
290     //下面是读取指令流:objfn.instrStream.datas
291     #define READ_BYTE()  (*ip++)    //从指令流中读取 1 字节
292     //读取指令流中的 2 字节
293     #define READ_SHORT() (ip += 2, (uint16_t)((ip[-2] << 8) | ip[-1]))
294
295     //当前指令单元执行的进度就是在指令流中的指针,即 ip,将其保存起来
296     #define STORE_CUR_FRAME() curFrame->ip = ip    // 备份 ip 以能回到当前
297
298     //加载最新的 frame
299     #define LOAD_CUR_FRAME()        \
300         /* frames 是数组,索引从 0 起,故 usedFrameNum-1 */    \
301         curFrame = &curThread->frames[curThread->usedFrameNum - 1]; \
302         stackStart = curFrame->stackStart; \
303         ip = curFrame->ip; \
304         fn = curFrame->closure->fn;
305
306     #define DECODE loopStart: \
307         opCode = READ_BYTE();\
308         switch (opCode)
309
310     #define CASE(shortOpCode) case OPCODE_##shortOpCode
311     #define LOOP() goto loopStart
```

函数 executeInstruction 用于执行当前线程 curThread 中的指令，即那些我们在语义分析过程中所生成的指令。

先把虚拟机 vm 中的 curThread 指向形参 curThread，以表示当前正在执行的线程，说到这儿大家一定明白了，线程切换时，我们始终是用 VM->curThread 指向当前运行的线程，一会就能看见其实现。接下来定义几个变量，curFrame 表示当前堆栈框架。stackStart 表示当前运行时栈的起始地址。ip 就是程序计数器，表示下一条指令在所运行的指令流中的地址。fn 表示当前运行的函数，即指令流。opCode 表示待执行的操作码。

前面我们说过，线程中的“大栈”被在其中运行的所有闭包函数的运行时栈共享（确切地说是“分”享，即各自分出一块作为自己的运行时栈，各分块并不重合），并且各闭包运行时栈



相互接壤。“大栈”的栈底是 ObjThread->stack，栈顶是 ObjThread->esp，而线程中各个闭包函数自己的运行时栈的栈底是 stackStart，它记录的是本运行时栈在“大栈”中的起始地址，因为任意时刻当前闭包函数的运行时栈都是位于线程中“大栈”的顶部，因此当前闭包函数的运行时栈的栈顶肯定是 ObjThread->esp。运行一个闭包函数就要为其准备一个堆栈框架 frame，在 frame 中记录了有关本闭包函数的运行环境。这是在之前介绍过的函数 prepareFrame 中设置的，摘录主要代码如下。

```
Frame* frame = &(objThread->frames[objThread->usedFrameNum++]);
frame->stackStart = stackStart;
frame->closure = objClosure;
frame->ip = objClosure->fn->instrStream.datas;
```

第 284～288 行定义的宏用于操作当前闭包函数的运行时栈，PUSH 用于把参数 value 入栈。POP 用于把栈顶元素（Value 结构）出栈，即获得栈顶元素。DROP 用于丢掉栈顶元素。PEEK 用于获得栈顶的数据，与 POP 不同的是，PEEK 并不改变栈顶指针 esp。PEEK2 用于获得次栈顶元素，即紧邻栈顶上面的元素，同样也不改变栈顶指针 esp。

接下来是定义了读取指令流的宏。READ_BYTE 用于从指令流中读取 1 字节，也就是返回 ip 的值并使 ip 自加 1。参数是以大端字节序写入 objfn.instrStream.datas 中的，即参数的高字节在低地址，参数的低字节在高地址。原因是数组元素地址是往高地址发展，大端更直观。宏 READ_SHORT 用于读取指令流中的 2 字节，2 字节的数据在写入时我们也是分拆两次写入的，即通过函数 writeShortOperand 实现的，现在用 READ_SHORT 将其还原。

堆栈框架就是函数的执行环境，每调用一个函数就要为其准备一个堆栈框架，函数闭包都关联到一个 frame。对于指令单元（如函数和方法）来说，其身份主要就是堆栈框架，函数调用也是从当前函数切换到另一函数，也就是切换堆栈框架。

下面的宏 STORE_CUR_FRAME 和 LOAD_CUR_FRAME 用于指令单元（如函数、方法）的堆栈框架的切换。宏 STORE_CUR_FRAME 用于存储当前 frame，切换或调用函数时就要保存相关 frame 中的运行环境，这有点类似于 CPU 在切换任务时，切换上下任务的上下文环境之前先把当前任务的上下文保存到 PCB（即程序控制块，现代操作系统会把任务的上下文执行环境保存到此结构中）中一样。STORE_CUR_FRAME 的实质就是把 ip 保存在 curFrame->ip，下次换该函数运行时再重新加载到 ip。

宏 LOAD_CUR_FRAME 用于加载最新的 frame。第 usedFrameNum-1 个 frame 是当前最新可用的堆栈框架，相当于最新的函数闭包。本宏中就是初始化上面定义的几个变量。提醒一下，curFrame->ip 是在函数 prepareFrame 中指向指令流的起始，上面已经列出了相关代码。

下面的宏 DECODE 用于解码，虚拟机就是个解码器，需要不断地解析指令流中的操作码及操作数，每一条指令都要有“解码、译码、执行”3 个阶段。“解码”相当于用 READ_BYTE 获取操作码到 opCode，“译码”相当于下面的 switch 分支，“执行”相当于 switch 中各 case 的实现。不过 DECODE 的值是个标号，即 loopStart:，这是虚拟机提速的关键，我们后面在执行一条指令后会直接 goto 到此标号以减少 CPU 跳出各分支的消耗。

宏 CASE 就是用在 switch 中的 case，可以看到在其实现中我们人为在操作码参数 shortOpcode 前补充了前缀 OPCODE_，这样调用 CASE 时传入的实参便可省去 OPCODE_，比如我们在后面都是这样用：CASE(LOAD_LOCAL_VAR)，比较省事。

宏 LOOP 就是 goto loopStart，跳到解码的开头。

7.4.2　解码、译码、执行（本节源码 stepByStep/c7/d）

各指令的操作码和操作数都是在编译阶段准备好的，编译和执行肯定是配合在一起的，大家



可以在 compiler.c 中对照各操作码及操作数的生成过程。操作码有 63 个，虚拟机要对每个操作码分别处理，为了简洁，在不至于混淆的情况下对于操作码我们就不说前缀 OPCODE 了。

1. 第一部分操作码

本节介绍的代码与上节的基础工作都是在同一个 vm.c 中，该文件太长了，因此拆分成 8 部分，现在是 vm.c 的第二部分，但它是“取指、译码、执行”的第一部分。

stepByStep/c7/d/vm/vm.c (2/8)

```
312
313     LOAD_CUR_FRAME();
314     DECODE {
315       //若 OPCODE 依赖于指令环境(栈和指令流),会在各 OPCODE 下说明
316
317       CASE(LOAD_LOCAL_VAR):
318         //指令流: 1 字节的局部变量索引
319
320         PUSH(stackStart[READ_BYTE()]);
321         LOOP();
322
323       CASE(LOAD_THIS_FIELD): {
324         //指令流: 1 字节的 field 索引
325
326         uint8_t fieldIdx = READ_BYTE();
327
328         //stackStart[0]是实例对象 this
329         ASSERT(VALUE_IS_OBJINSTANCE(stackStart[0]), "method receiver should be
            objInstance.");
330         ObjInstance* objInstance = VALUE_TO_OBJINSTANCE(stackStart[0]);
331
332         ASSERT(fieldIdx < objInstance->objHeader.class->fieldNum, "out of bounds
            field!");
333         PUSH(objInstance->fields[fieldIdx]);
334         LOOP();
335       }
336
337       CASE(POP):
338         DROP();
339         LOOP();
340
341       CASE(PUSH_NULL):
342         PUSH(VT_TO_VALUE(VT_NULL));
343         LOOP();
344
345       CASE(PUSH_FALSE):
346         PUSH(VT_TO_VALUE(VT_FALSE));
347         LOOP();
348
349       CASE(PUSH_TRUE):
```

```
350         PUSH(VT_TO_VALUE(VT_TRUE));
351         LOOP();
352
353     CASE(STORE_LOCAL_VAR):
354         //栈顶：局部变量值
355         //指令流：1 字节的局部变量索引
356
357         //将 PEEK()得到的栈顶数据写入指令参数(即 READ_BYTE()得到的值)为索引的栈的 slot 中
358         stackStart[READ_BYTE()] = PEEK();
359         LOOP();
360
361     CASE(LOAD_CONSTANT):
362         //指令流：2 字节的常量索引
363
364         //加载常量就是把常量表中的数据入栈
365         PUSH(fn->constants.datas[READ_SHORT()]);
366         LOOP();
```

操作码 LOAD_LOCAL_VAR 用于加载局部变量到栈顶，因为我们是基于栈的虚拟机，数据交换的地方都是在栈中，所以一般以 LOAD 开头的操作码都会是把操作数加载到栈顶（也只能放在栈顶，不可能插入到栈中）。局部变量是临时的，因此是存储在栈中的，我们需要到运行时栈中获取局部变量。此操作码的操作数是局部变量的索引，1 字节大小，位于指令流中，通过 READ_BYTE 来获取该索引，然后用此索引在运行时栈 startStack 中获取相应的局部变量。入栈操作是通过 PUSH 实现的，也就是加载到栈顶。到此完成了此指令的执行，随后调用 LOOP 处理下一个指令。

操作码 LOAD_THIS_FIELD 加载域到栈顶，操作数是位于指令流中域的索引，先通过 READ_BYTE 从指令流中读取索引到 fieldIdx。既然是加载 this 的域，那么位于栈底 stackStart[0] 的应该是 this 对象，第 330 行获取 this 对象到 objInstance。实例域是存储在实例对象中的 fields 数组中，我们在编译时也是把域写到该数组中的。因此通过 PUSH 把 objInstance->fields[fieldIdx] 压栈，至此完成了此指令的执行。最后调用 LOOP 继续处理下一条指令。接下来的 POP、PUSH_NULL、PUSH_FALSE 和 PUSH_TRUE 都很简单，不说了。

STORE_LOCAL_VAR 是把栈顶元素存储到局部变量中，操作数是栈顶元素和位于指令流中的局部变量的索引。通过 PEEK 得到栈顶的值后写入到运行时栈中局部变量的相应位置，调用 LOOP 继续循环。

LOAD_CONSTANT 就是把操作数表示的常量加载到栈顶，操作数是常量表中的索引，因此调用 READ_SHORT 获取常量索引后，从常量表 fn->constants.datas 中获取常量，最后通过 PUSH 把常量入栈。之后 LOOP 继续处理下一条指令。

2. 第二部分操作码

stepByStep/c7/d/vm/vm.c (3/8)

```
367
368     {
369         int argNum, index;
370         Value* args;
371         Class* class;
372         Method* method;
```

```
373
374         CASE(CALL0):
375         CASE(CALL1):
376         CASE(CALL2):
377         CASE(CALL3):
378         CASE(CALL4):
379         CASE(CALL5):
380         CASE(CALL6):
381         CASE(CALL7):
382         CASE(CALL8):
383         CASE(CALL9):
384         CASE(CALL10):
385         CASE(CALL11):
386         CASE(CALL12):
387         CASE(CALL13):
388         CASE(CALL14):
389         CASE(CALL15):
390         CASE(CALL16):
391             //指令流 1: 2 字节的 method 索引
392             //因为还有个隐式的 receiver(就是下面的 args[0]), 所以参数个数+1
393             argNum = opCode - OPCODE_CALL0 + 1;
394
395             //读取 2 字节的数据(CALL 指令的操作数),index 是方法名的索引
396             index = READ_SHORT();
397
398             //为参数指针数组 args 赋值
399             args = curThread->esp - argNum;
400
401             //获得方法所在的类
402             class = getClassOfObj(vm, args[0]);
403             goto invokeMethod;
404
405         CASE(SUPER0):
406         CASE(SUPER1):
407         CASE(SUPER2):
408         CASE(SUPER3):
409         CASE(SUPER4):
410         CASE(SUPER5):
411         CASE(SUPER6):
412         CASE(SUPER7):
413         CASE(SUPER8):
414         CASE(SUPER9):
415         CASE(SUPER10):
416         CASE(SUPER11):
417         CASE(SUPER12):
418         CASE(SUPER13):
419         CASE(SUPER14):
420         CASE(SUPER15):
421         CASE(SUPER16):
422             //指令流 1: 2 字节的 method 索引
```

```
423         //指令流 2: 2 字节的基类常量索引
424
425         //因为还有个隐式的 receiver(就是下面的 args[0]), 所以参数个数+1
426         argNum = opCode - OPCODE_SUPER0 + 1;
427         index = READ_SHORT();
428         args = curThread->esp - argNum;
429
430         //在函数 bindMethodAndPatch 中实现的基类的绑定
431         class = VALUE_TO_CLASS(fn->constants.datas[READ_SHORT()]);
432
433      invokeMethod:
434         if ((uint32_t)index > class->methods.count ||
435                     (method = &class->methods.datas[index])->type == MT_NONE) {
436            RUN_ERROR("method \"%s\" not found!", vm->allMethodNames.datas[index].str);
437         }
438
439         switch (method->type) {
440            case MT_PRIMITIVE:
441
442               //如果返回值为 true,则 vm 进行空间回收的工作
443               if (method->primFn(vm, args)) {
444                  //args[0]是返回值, argNum-1 是保留 args[0],
445                  //args[0]的空间最终由返回值的接收者即函数的主调方回收
446                  curThread->esp -= argNum - 1;
447               } else {
448                  //如果返回 false 则说明有两种情况:
449                  //   1. 出错(比如原生函数 primThreadAbort 使线程报错或无错退出),
450                  //   2. 或者切换了线程,此时 vm->curThread 已经被切换为新的线程
451                  //保存线程的上下文环境,运行新线程之后还能回到当前老线程指令流的正确位置
452                  STORE_CUR_FRAME();
453
454                  //如果没有待执行的线程,说明执行完毕
455                  if (vm->curThread == NULL) {
456                      return VM_RESULT_SUCCESS;
457                  }
458
459                  //vm->curThread 已经由返回 false 的函数置为下一个线程
460                  //切换到下一个线程的上下文
461                  curThread = vm->curThread;
462                  LOAD_CUR_FRAME();
463
454                  if (!VALUE_IS_NULL(curThread->errorObj)) {
455                       if (VALUE_IS_OBJSTR(curThread->errorObj)) {
456                            ObjString* err = VALUE_TO_OBJSTR(curThread->errorObj);
457                            printf("%s", err->value.start);
458                       }
459                       //出错后将返回值置为 null,避免主调方获取到错误的结果
460                       PEEK() = VT_TO_VALUE(VT_NULL);
461                  }
462
463                  //如果没有待执行的线程,说明执行完毕
```

```
464                 if (vm->curThread == NULL) {
465                     return VM_RESULT_SUCCESS;
466                 }
467
468                 //vm->curThread 已经由返回 false 的函数置为下一个线程
469                 //切换到下一个线程的上下文
470                 curThread = vm->curThread;
471                 LOAD_CUR_FRAME();
472             }
473             break;
474
475         case MT_SCRIPT:
476             STORE_CUR_FRAME();
477             createFrame(vm, curThread, (ObjClosure*)method->obj, argNum);
478             LOAD_CUR_FRAME();   //加载最新的 frame
479             break;
480
481         case MT_FN_CALL:
482             ASSERT(VALUE_IS_OBJCLOSURE(args[0]), "instance must be a closure!");
483             ObjFn* objFn = VALUE_TO_OBJCLOSURE(args[0])->fn;
484             //-1 是去掉实例 this
485             if (argNum - 1 < objFn->argNum) {
486                 RUN_ERROR("arguments less");
487             }
488
489             STORE_CUR_FRAME();
490             createFrame(vm, curThread, VALUE_TO_OBJCLOSURE(args[0]), argNum);
491             LOAD_CUR_FRAME();   //加载最新的 frame
492             break;
493
494         default:
495             NOT_REACHED();
496     }
497
498     LOOP();
499 }
```

第 368 行的左大括号是为了在 case 中声明一些变量，index 是所调用方法的索引，argNum 用来表示所调用方法的参数个数，args 表示所调用方法的参数数组，class 表示方法所在的类，method 用来表示所调用方法的方法体。

接下来的是操作码 CALL0～CALL16，这些操作码是用来调用参数个数在 0～16 的方法，即 CALL0 调用无参数的方法，CALL16 调用 16 个参数的方法，我们的方法最多支持 16 个参数。这些操作码全部统一处理，流程直奔第 393 行。下面依次获取相关的值到以上声明的变量中，argNum 是由操作码减去 OPCODE_CALL0 再加 1 得到的，index 是调用 READ_SHORT 从指令流中获取，调用方法前都提前把参数加载到了栈顶，因此第 0 个参数的地址可以用 esp 减 argNum 获得，第 399 行的 args 直接指向栈中的参数。第 402 行是对 class 的赋值，通过 getClassOfObj 获得它所属的类。args[0] 是消息的接收者，其值有两种可能，如果当前 CALLx 调用的是类方法即 static 方法，那么 args[0] 是类，因此变量 class 便指向静态方法所属类的 Meta 类。如果当前调用的是实例方法，那么 args[0] 是实例对象 this，因此变量 class 便是对象实例所属的类。第 403 行跳到标号 invokeMethod 去调用

仅供非商业用途或交流学习使用

方法，method 变量在该标号中才用上。在这之前先看下基类指令 SUPER0～SUPER16，这些指令是用来调用基类中的方法，同 CALL0～16 一样，参数个数是 0～16。基类的索引也是位于指令流中，它是作为常量存在，因此从常量表中获取该基类，转换为类后存储到 class。

接下来就是标号 invokeMethod 了，第 435 行从类中获取方法到 method 中，如果调用的是不存在的方法，此时并不报错，因为这是一遍的编译器，很多东西都是可以先引用后定义，因此只有在编译完整代码后我们才能做检查，但我们并没有这样做，而是在运行时判断所调用的方法是否存在。下面根据方法的类型分别处理，第 440 行处理原生方法的调用，第 443 行调用原生方法，我们在之前已经实现了部分原生方法，它们返回 bool 值。如果返回 true，说明原生方法执行正常，并且需要由主调方回收参数的空间，因此第 446 行使 esp 减去 argNum 与 1 的差以实现参数的空间回收。减 1 的目的是避免回收 args[0]，因为被调用的方法用 args[0]来存储返回值，并且由于主调方和被调方的运行时栈接壤，所以主调方才能在自己的栈顶（即此处的 args[0]）获取到被调用方法的结果。说明一下，args[0]所在的 slot 就是 stackStart[0]，即本函数或方法运行时栈的起始。如果原生方法返回 false，那么虚拟机的线程调度器就会进行线程切换，即调度下一个线程上 CPU。这是我们设计好的约定——线程与虚拟机线程调度器之间的调度、切换约定：返回 false 就切换线程。所以，原生方法返回了 false 只有两种可能，要么方法执行中捕捉到某种错误，无法运行下去了，因此需要交出 CPU 使用权，让下一个线程运行，于是主动 return false；要么就是被调用的方法使 VM->curThread 指向为下一个要切换的线程后，主动调用了 Thread. yield()或 Thread. yield(arg)方法（这两个方法的最后都是 return false）要求线程切换。表面上是两种情况，其实本质上就是一种情况，就是切换线程，这是线程调度。线程是在用户态实现的，因此也称为协程。协程由进程自己安排调度，此处的代码就是协程的调度器。第 452 行调用 STORE_CUR_FRAME 保存当前线程的上下文环境。接下来判断 curThread->errorObj 是否为空，如果不为空则说明线程出错了，当前的切换是由于出错导致的，因此打印出报错，并将返回值置为 NULL。第 464 行判断若 VM->curThread 为 NULL，说明没有待切换的线程，因此虚拟机执行完毕。目前这里是一律返回 VM_RESULT_SUCCESS，对于运行出错的情况我们会及时报错，以后可以改为异常处理。VM->curThread 已经由返回 false 的方法置为下一个可执行的线程，因此第 470 行更新 curThread 为 VM->curThread，下面调用 LOAD_CUR_FRAME 加载新线程的上下文环境。

第 475 行处理脚本方法(类型为 MT_SCRIPT)的调用，这个就很简单了，先调用 STORE_CUR_FRAME 保存上下文，然后调用 createFrame 为方法创建堆栈框架并使方法的堆栈框架就绪（准备运行），最后调用 LOAD_CUR_FRAME 加载该方法的堆栈框架使方法上 CPU 运行。

第 481 行处理函数调用，类型为 MT_FN_CALL，method.type 为 MT_FN_CALL 时 method.obj 和 method.primFn 为空，因为实例本身就是待调用的函数，以后我们还会在 core.c 中完善原生方法，到时候大家就明白了。所以第 483 行调用 VALUE_TO_OBJCLOSURE 将其转换为 objFn。后面和前面的两种类型也差不多，不多说了。

也许有读者会问，切换线程只发生在原生方法返回 false 时，那么脚本方法就不能切换线程吗？原因是，用户是不知道脚本中哪些方法是原生方法的，线程切换的方法 Thread.yield()本身就是原生方法，因此脚本中的代码调用该方法时必然会执行上面 case MT_PRIMITIVE 的流程。

3. 第三部分操作码

stepByStep/c7/d/vm/vm.c (4/8)

```
500
501         CASE(LOAD_UPVALUE):
502             //指令流：1 字节的 upvalue 索引
```

仅供非商业用途或交流学习使用


```
503             PUSH(*((curFrame->closure->upvalues[READ_BYTE()])->localVarPtr));
504             LOOP();
505
506         CASE(STORE_UPVALUE):
507             //栈顶：upvalue 值
508             //指令流：1 字节的 upvalue 索引
509
510             *((curFrame->closure->upvalues[READ_BYTE()])->localVarPtr) = PEEK();
511             LOOP();
512
513         CASE(LOAD_MODULE_VAR):
514             //指令流：2 字节的模块变量索引
515
516             PUSH(fn->module->moduleVarValue.datas[READ_SHORT()]);
517             LOOP();
518
519         CASE(STORE_MODULE_VAR):
520             //栈顶：模块变量值
521
522             fn->module->moduleVarValue.datas[READ_SHORT()] = PEEK();
523             LOOP();
524
525         CASE(STORE_THIS_FIELD): {
526             //栈顶：field 值
527             //指令流：1 字节的 field 索引
528
529             uint8_t fieldIdx = READ_BYTE();
530             ASSERT(VALUE_IS_OBJINSTANCE(stackStart[0]), "receiver should be instance!");
531                 ObjInstance* objInstance = VALUE_TO_OBJINSTANCE(stackStart[0]);
532             ASSERT(fieldIdx < objInstance->objHeader.class->fieldNum,
              "out of bounds field!");
533             objInstance->fields[fieldIdx] = PEEK();
534             LOOP();
535         }
536
537         CASE(LOAD_FIELD): {
538             //栈顶:实例对象
539             //指令流：1 字节的 field 索引
540
541             uint8_t fieldIdx = READ_BYTE();  //获取待加载的字段索引
542             Value receiver = POP();   //获取消息接收者
543             ASSERT(VALUE_IS_OBJINSTANCE(receiver), "receiver should be instance!");
544             ObjInstance* objInstance = VALUE_TO_OBJINSTANCE(receiver);
545             ASSERT(fieldIdx < objInstance->objHeader.class->fieldNum,
              "out of bounds field!");
546             PUSH(objInstance->fields[fieldIdx]);
547             LOOP();
548         }
549
550         CASE(STORE_FIELD): {
```

仅供非商业用途或交流学习使用


```
551          //栈顶:实例对象 次栈顶:field 值
552          //指令流: 1 字节的 field 索引
553
554          uint8_t fieldIdx = READ_BYTE();  //获取待加载的字段索引
555          Value receiver = POP();    //获取消息接收者
556          ASSERT(VALUE_IS_OBJINSTANCE(receiver), "receiver should be instance!");
557          ObjInstance* objInstance = VALUE_TO_OBJINSTANCE(receiver);
558          ASSERT(fieldIdx < objInstance->objHeader.class->fieldNum,
             "out of bounds field!");
559          objInstance->fields[fieldIdx] = PEEK();
560          LOOP();
561       }
```

第 501 行的操作码 LOAD_UPVALUE 用于处于 upvalue 的加载。指令流中是 upvalue 的索引，通过该索引从 upvalues 数组中获取到相应的 upvalue，调用 PUSH 将其压栈，然后 LOOP 处理下一个指令。

下面是处理操作码 STORE_UPVALUE，它用于保存 upvalue。栈顶是 upvalue 的值，指令流是 upvalue 的索引，第 510 行将其保存。继续 LOOP。

后面的操作码 LOAD_MODULE_VAR 和 STORE_MODULE_VAR 分别用于加载（读取）和存储模块变量。

操作码 STORE_THIS_FIELD 用于存储 this 对象的域，域值位于栈顶，域索引位于指令流。运行时栈第 0 个 slot，即 stackStart[0]的值是实例对象，域存储在对象的 fields 数组中，因此第 533 行通过 PEEK 从栈顶获取域值后写入 objInstance->fields[fieldIdx]完成域的存储。

第 537 行的操作码 LOAD_FIELD 用于加载域到栈顶，此次并不是加载 this 对象的域，因此并不是从 stackStart[0]中获取对象实例，而是从栈顶中获取对象实例，编译时已经生成将对象实例压栈的指令，而在这之前已经执行过它了。指令流中的是域的索引。原理是先从指令流中获取域的索引值存入 fieldIdx，然后从栈顶中获取对象的 Value 结构存入 receiver，然后将其转换为实例对象存入 objInstance。然后通过域的索引从该对象中的 fields 数组中获取到域后压栈，即第 546 行的代码。

操作码 STORE_FIELD 用于存储域值。栈顶是实例对象，次栈顶是域值，指令流中的是域的索引。以下先后获取到域索引 fieldIdx，对象实例 objInstance，然后第 559 行把域值写入对象中相应的域。

4. 第四部分操作码

stepByStep/c7/d/vm/vm.c (5/8)

```
562
563       CASE(JUMP): {
564          //指令流: 2 字节的跳转正偏移量
565
566          int16_t offset = READ_SHORT();
567          ASSERT(offset > 0, "OPCODE_JUMP's operand must be positive!");
568          ip += offset;
569          LOOP();
570       }
571
572       CASE(LOOP): {
```

仅供非商业用途或交流学习使用


```
573             //指令流：2 字节的跳转正偏移量
574
575             int16_t offset = READ_SHORT();
576                 ASSERT(offset > 0, "OPCODE_LOOP's operand must be positive!");
577             ip -= offset;
578             LOOP();
579         }
580
581         CASE(JUMP_IF_FALSE): {
582             //栈顶：跳转条件 bool 值
583             //指令流：2 字节的跳转偏移量
584
585             int16_t offset = READ_SHORT();
586             ASSERT(offset > 0, "OPCODE_JUMP_IF_FALSE's operand must be positive!");
587             Value condition = POP();
588             if (VALUE_IS_FALSE(condition) || VALUE_IS_NULL(condition)) {
589                 ip += offset;
590             }
591             LOOP();
592         }
593
594         CASE(AND): {
595             //栈顶：跳转条件 bool 值
596             //指令流：2 字节的跳转偏移量
597
598             int16_t offset = READ_SHORT();
599             ASSERT(offset > 0, "OPCODE_AND's operand must be positive!");
600             Value condition = PEEK();
601
602             if (VALUE_IS_FALSE(condition) || VALUE_IS_NULL(condition)) {
603                 //若条件为假则不再计算 and 的右操作数，跳过右操作数的计算指令
604                 ip += offset;
605             } else {
606                 //若条件为真则继续执行 and 右边的表达式计算步骤，丢掉栈顶的条件
607                 DROP();
608             }
609             LOOP();
610         }
611
612         CASE(OR): {
613             //栈顶：跳转条件 bool 值
614             //指令流：2 字节的跳转偏移量
615
616             int16_t offset = READ_SHORT();
617             ASSERT(offset > 0, "OPCODE_OR's operand must be positive!");
618             Value condition = PEEK();
619
620             if (VALUE_IS_FALSE(condition) || VALUE_IS_NULL(condition)) {
621                 //若条件为假或空则执行 or 右操作数的计算步骤，丢掉跳转条件
622                 DROP();
```

仅供非商业用途或交流学习使用


```
623             } else {
624                 //若条件为真则跳过 or 右边的表达式,无须计算
625                  ip += offset;
626             }
627             LOOP();
628         }
```

第 563 行操作码 JUMP 用于无条件向前跳转，指令流中的操作数是 2 字节大小的偏移量，偏移量都是正数，跳转原理即是使 ip 加上偏移量 offset 即可。

操作码 LOOP 用于无条件向回跳转，跳转偏移量也在指令流中，偏移量都为正数，因此向回跳转是使 ip 减去偏移量。

操作码 JUMP_IF_FALSE 用于条件失败时跳转，条件位于栈顶，跳转偏移量位于指令流中。对于条件来说，有两种情况为失败：false 和 null。如果满足两者之一，使 ip 加上偏移量 offset 实现跳转。说明一下，0 在我们语言中并不为假，反而为真，在很多语言中都是哪些，比如 ruby。其实这是有道理的，这主要体现在迭代器中，比如如下代码。

```
[work@work spr]$ cat -n /tmp/test.sp
     1var seq = [1,2,3,4,5]
     2var iter
     3while (iter = seq.iterate(iter)) {
     4    var i = seq.iteratorValue(iter)
     5    System.print(i)
     6}
[work@work spr]$ ./spr /tmp/test.sp
1
2
3
4
5
[work@work spr]$
```

以上代码是循环打印列表中所有元素。第 2 行变量 iter 的值为 null，那么传给第 3 行的 iterate 做参数时，iterate 对于 null 值的迭代器返回 0，此时 0 又作为循环条件，如果 0 为假的话，那就无法实现遍历了，所以 0 为真值是合理的。

操作码 AND 表示逻辑与运算。栈顶是左操作数，即跳转条件，如 A && B 中的 A。指令流中的是跳转偏移量，即用于计算右操作数的一系列指令的长度。也就是说当左操作数为假时，要跳过右操作数（计算右操作数的一系列指令）。第 602 行判断条件是否为假，若为假就使 ip 加上偏移量实现跨过右操作数的计算。也就是说，如果不跳的话，此时顺着 ip 向下执行便是计算右操作数的一系列指令，这些指令是由函数 logicAnd 中调用 expression(cu, BP_LOGIC_AND)生成的。如果条件为真，也就是说左操作数为真，那么调用 DROP 回收跳转条件所在的栈顶空间，即丢弃了条件（即 A&&B 中的 A），执行 LOOP 以继续执行计算右操作数的指令。

操作码 OR 用于逻辑或运算。栈顶是左操作数，即跳转条件，指令流中的是跳转偏移量，即用于计算右操作数的指令的长度。当左操作数为真时，就要跳过计算右操作数的一系列指令。

第 620 行判断条件是否为假，若为假就要继续执行计算右操作数的指令。然后回收跳转条件所在的栈顶空间，即丢弃了条件（即 A||B 中的 A 的结果）。此时顺着 ip 向下执行便是函数 logicOr 中 expression(cu, BP_LOGIC_OR)所生成的计算表达式'B'的一系列指令。第 623 行若条件为真，右操作数就不需要计算了，因此使 ip 加上偏移量 offset 实现跳过。

仅供非商业用途或交流学习使用

5. 第五部分操作码

stepByStep/c7/d/vm/vm.c (6/8)

```
629
630         CASE(CLOSE_UPVALUE):
631             //栈顶：相当于局部变量
632             //把地址大于栈顶局部变量的 upvalue 关闭
633             closedUpvalue(curThread, curThread->esp - 1);
634             DROP();    //弹出栈顶局部变量
635             LOOP();
636
637         CASE(RETURN): {
638             //栈顶：返回值
639
640             //获取返回值
641             Value retVal = POP();
642
643             //return 是从函数返回，故该堆栈框架使用完毕，增加可用堆栈框架数量
644             curThread->usedFrameNum--;
645
646             //关闭堆栈框架即此作用域内所有 upvalue
647             closedUpvalue(curThread, stackStart);
648
649             //如果一个堆栈框架都没用，
650             //说明它没有调用函数或者所有的函数调用都返回了，可以结束它
651             if (curThread->usedFrameNum == 0) {
652                 //如果并不是被另一线程调用的，就直接结束
653                 if (curThread->caller == NULL) {
654                     curThread->stack[0] = retVal;
655
656                     //保留 stack[0]中的结果，其他都丢弃
657                     curThread->esp = curThread->stack + 1;
658                     return VM_RESULT_SUCCESS;
659                 }
660
661                 //恢复主调方线程的调度
662                 ObjThread* callerThread = curThread->caller;
663                 curThread->caller = NULL;
664                 curThread = callerThread;
665                 vm->curThread = callerThread;
666
667                 //在主调线程的栈顶存储被调线程的执行结果
668                 curThread->esp[-1] = retVal;
669             } else {
670                 //将返回值置于运行时栈栈顶
671                 stackStart[0] = retVal;
672                 //回收堆栈：保留除结果所在的 slot 即 stackStart[0]，其他全丢弃
673                 curThread->esp = stackStart + 1;
674             }
```

仅供非商业用途或交流学习使用


```
675
676             LOAD_CUR_FRAME();
677             LOOP();
678         }
679
680         CASE(CONSTRUCT): {
681             //栈底: startStart[0]是 class
682
683             ASSERT(VALUE_IS_CLASS(stackStart[0]),
684                                 "stackStart[0] should be a class for OPCODE_CONSTRUCT!");
685
686             //将创建的类实例存储到 stackStart[0],即 this
687             ObjInstance* objInstance = newObjInstance(vm,
                VALUE_TO_CLASS(stackStart[0]));
688             //此时 stackStart[0]是类,其类名便是方法所定义的类
689             //把对象写入 stackStart[0]
690             stackStart[0] = OBJ_TO_VALUE(objInstance);
691
692             LOOP();
693         }
```

操作码 CLOSE_UPVALUE 用于关闭 upvalue，这分为两步。

（1）先调用 closedUpvalue，把大于栈顶的 upvalue 关闭，大家要理解 closedUpvalue 的意义，这其实是在做善后工作，是对内层函数使用本运行时栈中局部变量的妥善处理，即让那些把本运行时栈中局部变量作为 upvalue 的函数或方法把该局部变量的值存储到 ObjUpvalue->closedUpvalue 中，然后使 localVarPtr 指向 ObjUpvalue->closedUpvalue。

（2）通过 DROP 丢掉栈顶元素。

操作码 RETURN 是脚本中 return 语句对应的指令，它结束一个函数的运行，并且把栈顶的值作为 return 语句的返回值。运行一个函数就需要一个 frame，因此操作码 RETURN 会结束一个 frame。先通过 POP 从栈顶获取返回值到 retVal，然后使线程 curThread->usedFrameNum 自减 1，然后调用 closedUpvalue 关闭本运行时栈中的 upvalue。第 651 行判断若 usedFrameNum 减 1 后为 0，这说明本线程一个堆栈框架都没用上，说明它已不再执行函数或者说所有执行的函数调用都返回了，可以结束本线程了。线程可以由另一个线程调用，也就是说线程也可以有主调方，这是由 curThread->caller 来记录的，如果其为 NULL，说明本线程并不是由另一线程调用才运行的，因此把返回值 retVal 放在栈底即 stack[0]，然后回收“大栈”中除 stack[0]之外的空间，即代码 curThread->esp = curThread->stack + 1，最后返回 VM_RESULT_SUCCESS 宣告虚拟机成功结束。到了第 661 行，说明有主调方线程，那么就将控制权转交给主调方，调用一个线程的时候会在被调线程的 caller 成员中记录主调方线程，因此第 662 行获取主调方线程 callerThread，然后置空被调方线程的 caller，即使其为 NULL，然后在第 664 行设置表示当前线程的变量 curThread 为主调方线程 callerThread，然后更新虚拟机中的当前线程 vm->curThread 为 callerThread，使主调方线程就绪。最后在主调方线程的栈顶存储返回值，即代码 curThread->esp[-1] = retVal。第 669 行，如果 curThread->usedFrameNum 不为 0，说明当前线程在层层的函数调用链中尚未返回到最上层函数，那么需要将返回值放到本函数的运行时栈的首地址 stackStart[0]，这样主调方法或函数才能在它的运行时栈的栈顶获取到该返回值。然后回收运行时栈中除返回值所在的 slot（stackStart[0]），即代码 curThread->esp = stackStart + 1 的工作。注意，此行代码回收的空间包括函数参数的空间，

仅供非商业用途或交流学习使用

这相当于是由被调函数自己在回收的，与原生方法中主调方负责回收不一样。最后调用 LOAD_CUR_FRAME，恢复主调方函数或方法的运行。

操作码 CONSTRUCT 用于创建对象，此时运行时栈的栈底 startStart[0]是 class，这是由操作码 CREATE_CLASS 创建的，也就是说在执行 CONSTRUCT 之前必须执行过了 CREATE_CLASS。接着调用 newObjInstance 创建对象实例，将对象实例转换为 Value 结构后存储在 startStack[0]。

最后我再强调一下两种方法回收参数空间的区别，对于原生方法来说，方法的参数是由主调方回收的，在 case MT_PRIMITIVE:中，原生方法返回 true 后，由代码 curThread->esp -= argNum - 1 负责回收。对于脚本方法来说，参数是由操作码 RETURN 回收的，由代码 curThread->esp = stackStart + 1 完成。

6. 梳理 upvalue

下面是创建闭包的操作码 CREATE_CLOSURE，在进一步介绍之前我们要梳理一下。

说到闭包就要说到 upvalue，有了 upvalue 这个自由变量才谈得上闭包。在本语言系统中，和 upvalue 相关的结构分为两大类，一类是在“编译阶段”，如局部变量 LocalVar 结构中的 isUpvalue 和编译单元中的 upvalues 数组，不过这些只是用于编译时统计 upvalue，是“假的 upvalue”，完全是为生成闭包中“真正的 upvalue”做准备。而这个“真正的 upvalue”就是我们所说的另一类，也就是“运行时阶段”创建闭包时所创建的，即 ObjClosure->upvalues 数组元素中的 upvalue，这个是操作码 CREATE_CLOSURE 中的重点，先梳理编译阶段的“假 upvalue”。以下所说内容都是在编译阶段中。

LocalVar->isUpvalue 是在 findupvalue 函数中被赋值的，表示此局部变量是否是 upvalue。相关代码如下。

```
int directOuterLocalIndex = findLocal(cu->enclosingUnit, name, length);
if (directOuterLocalIndex != -1) {
    cu->enclosingUnit->localVars[directOuterLocalIndex].isUpvalue = true;
    return addUpvalue(cu, true, (uint32_t)directOuterLocalIndex);
}
```

LocalVar->isUpvalue 只会用在 discardLocalVar 函数中，用于判断若为 upvalue 就生成 OPCODE_CLOSE_UPVALUE 去关闭 upvalue，相关代码如下。

```
if (cu->localVars[localIdx].isUpvalue) {
   writeByte(cu, OPCODE_CLOSE_UPVALUE);
} else {
   writeByte(cu, OPCODE_POP);
}
```

而 compileUnit->upvalues 是 Upvalue 结构，结构中的成员是由 addUpvalue 统计的，此函数也是在 findUpvalue 函数中被调用。addUpvalue 中的相关代码如下。

```
cu->upvalues[cu->fn->upvalueNum].isEnclosingLocalVar = isEnclosingLocalVar;
cu->upvalues[cu->fn->upvalueNum].index = index;
return cu->fn-> upvalueNum++;
```

在编译完编译单元后，endCompileUnit 函数会利用上面收集的信息为创建闭包的操作码 CREATE_CLOSURE 提供操作数，相关代码如下。

```
index = 0;
```

仅供非商业用途或交流学习使用


```
while (index < cu->fn->upvalueNum) {
    writeByte(cu->enclosingUnit, cu->upvalues[index].isEnclosingLocalVar ? 1 : 0);
    writeByte(cu->enclosingUnit, cu->upvalues[index].index);
    index++;
}
```

以上所说的编译阶段的 upvalue 都只是跟踪、统计 upvalue 的信息，最终是为“运行时”阶段即虚拟机执行操作码 CREATE_CLOSURE 创建闭包时做准备的，闭包中的 upvalue 即 ObjClosure->upvalues 数组中的元素才是真正的 upvalue，毕竟 upvalue 是给闭包用的，必须用真的。

7. 第六部分操作码

闭包的概念其实已经让很多人迷惑了，何况我们要实现闭包，因此本小节有点难度，我会尽量解释清楚。

简单来说，闭包就是待封闭的函数+自由变量，自由变量就是 upvalue，函数是编译单元即 compileUnit。正因为编译单元可以是嵌套的，这才有了闭包的概念，闭包正是被嵌套于某编译单元中的编译单元。创建闭包的操作码 CREATE_CLOSURE 位于闭包所属的父（直接外层）编译单元中，这是由 endCompileUnit 函数在编译完某编译单元后装入其父编译单元中的，也就是说此时正在执行的操作码 CREATE_CLOSURE 是在闭包的直接外层编译单元的指令流中。函数或方法等都是编译单元，举个例子，最外层的函数 0 包含函数 1 的定义，函数 1 又包含函数 2 的定义，内层函数都是以闭包形式存在，既然创建闭包的指令肯定是在其直接外层编译单元的指令流中，也就是说只有在执行了父编译单元的指令流时才会遇到创建内层闭包的指令（操作码 OPCODE_CREATE_CLOSURE 及其操作数）。作用域规定外层函数不允许引用其内层作用域中的变量，因此 upvalue 指的就是引用其外层的局部变量 up、value，即上层编译单元中的值。函数 0 是最外层的，在它外面没有局部变量了，所以函数 0 中并没有 upvalue，但函数 0 中可以有自己的局部变量，此作用域内的局部变量可供其内部的作用域引用，即函数 0 中的局部变量可作为函数 1 和函数 2 的 upvalue。函数 1 的父编译单元就是函数 0，函数 1 中引用的 upvalue 必然只是函数 0 中的局部变量。假设当前执行的是函数 0 的指令流，此时执行的操作码 OPCODE_CREATE_CLOSURE 是在编译函数 1 结束时由函数 endCompileUnit 往其父编译单元（函数 0）中装入的，旨在父编译单元的指令流中添加创建闭包 1 的指令，即操作码 OPCODE_CREATE_ CLOSURE 及其操作数。

代码如下，后面会继续以此例子帮助大家理解。

stepByStep/c7/d/vm/vm.c (7/8)

```
694
695         CASE(CREATE_CLOSURE): {
696             //指令流: 2 字节待创建闭包的函数在常量表中的索引+函数所用的 upvalue 数 × 2
697
698             //endCompileUnit 已经将闭包函数添加进了常量表
699             ObjFn* objFn = VALUE_TO_OBJFN(fn->constants.datas[READ_SHORT()]);
700             ObjClosure* objClosure = newObjClosure(vm, objFn);
701             //将创建好的闭包的 value 结构压到栈顶,
702             //后续会有函数如 defineMethod 从栈底取出
703             //先将其压到栈中,后面再创建 upvalue,这样可避免在创建 upvalue 过程中被 GC
704             PUSH(OBJ_TO_VALUE(objClosure));
705
```

```
706             uint32_t idx = 0;
707             while (idx < objFn->upvalueNum) {
708                 //读入 endCompilerUnit 函数最后为每个 upvale 写入的数据对儿
709                 uint8_t isEnclosingLocalVar = READ_BYTE();
710                 uint8_t index = READ_BYTE();
711
712                 if (isEnclosingLocalVar) {    //是直接外层的局部变量
713                     //创建 upvalue
714                     objClosure->upvalues[idx] =
715                         createOpenUpvalue(vm, curThread, curFrame->stackStart + index);
716                 } else {
717                     //直接从父编译单元中继承
718                     objClosure->upvalues[idx] = curFrame->closure->upvalues[index];
719                 }
720                 idx++;
721             }
722
723             LOOP();
724         }
```

操作码 CREATE_CLOSURE 的操作数位于指令流，操作数包括 2 字节待创建闭包的函数在常量表中的索引，加上函数所用的 upvalue 数量的 2 倍，因此操作数的个数不定，这取决于 upvalue 的数量。提醒一下，我们在编译完编译单元后，调用函数 endCompileUnit 为每个 upvalue 在指令流中都创建了两个参数，分别是 upvalue 是否为直接外层编译单元的局部变量（即布尔值 endCompileUnit-> upvalues[xxx].isEnclosingLocalVar）和 upvalue 的索引（即 endCompileUnit-> upvalues[xxx].index），因此操作数要加上 upvalue 数量的 2 倍。

用于创建闭包的函数已经被 endCompileUnit 添加进了常量表，因此先调用 READ_SHORT 从指令流中获取常量索引，然后通过该索引从常量表中获取该函数后存储到 objFn。然后调用 newObjClosure 为 objFn 创建闭包，随后把闭包压入栈顶。闭包是由“函数+自由变量即 upvalue”组成的，此时只完成了函数的部分，下面创建 upvalue。upvalue 是由函数使用的，因此函数 objFn 所使用的 upvalue 的数量存储在 objFn->upvalueNum，下面通过 while 循环依次读取每个 upvalue 的两个操作数，即分别通过宏 READ_BYTE 获取布尔值 isEnclosingLocalVar 和索引 index。强调一下，布尔值 isEnclosingLocalVar 表示的是该 upvalue 是否为父编译单元的局部变量。若其为 true，则通过 createOpenUpvalue 创建 upvalue。

继续接本小节开头的例子，当 vm 执行函数 0 指令流中为函数 1 创建闭包 1 的指令时，函数 1 中 upvalue（即引用的函数 0 中的局部变量）的 isEnclosingLocalVar（即原来在编译阶段收集到的 upvalue 信息 compileUnit->upvalues[xxx].isEnclosingLocalVar）肯定为 true，因此会执行此处第 712 行的 true 分支，也就是调用 createOpenUpvalue 为闭包创建真正的 upvalue。我们知道，upvalue 是父编译单元（相当于函数 0）中的局部变量，局部变量位于运行时栈中，createOpenUpvalue 的形参 curFrame->stackStart + index 中的 curFrame->stackStart 是当前函数（即所创建闭包的父编译单元，此处相当于是函数 0）运行时栈的起始地址，index 是 upvalue 对应的局部变量在运行时栈中的索引，因此 curFrame->stackStart + index 就是 upvalue 所指向的局部变量的内存地址。提醒一下，通过 createOpenUpvalue 返回的 ObjUpvalue 是已加入到 ObjThread->openValues 链表中的节点，忘记的读者复习下函数 createOpenUpvalue，因此闭包 objClosure->upvalues 数组实际上是共享线程 curThread->openValues 这个全局的链表，也就是说

此链表被线程中所有闭包共享。

同理，函数 2 是在函数 1 的作用域内，在编译函数 2 后同样也会由 endCompileUnit 函数将 OPCODE_CREATE_CLOSURE 装入函数 1 的指令流中。当 vm 执行函数 1 指令流中的 OPCODE_CREATE_CLOSURE 时就去创建函数 2 的闭包 2，闭包 2 中引用的 upvalue 有可能是函数 1 中的局部变量，即此时 isEnclosingLocalVar 为 true，也可能是函数 1 中的 upvalue（即函数 1 所引用函数 0 中的局部变量），那么 isEnclosingLocalVar 为 false。当然，如果是第一种情况就会执行第 712 行的 true 分支，现在我们讨论的是第二种，也就是第 718 行的 else 分支。提醒一下，编译函数（如函数 2）的过程中查找所引用的变量时已经通过 findUpvalue 将 upvalue 添加到各中间层函数中了，比如函数 2 中用到的 upvalue 是函数 0 的局部变量 a，而函数 1 未必用到了函数 0 中的局部变量 a，findUpvalue 在为函数 2 向上递归查找到函数 0 时，也会把变量 a 放到中间层即函数 1 编译单元 compileUnit->upvalues 数组中，也就是说，无论最内层引用的 upvalue 指向了哪一个外层的局部变量，最内层与该外层间的所有中间层都会把该局部变量作为自己的 upvalue 存储到中间层自己的 compileUnit->upvalues 数组中。既然创建闭包的指令是在父编译单元中，而函数又可以层层嵌套定义，因此在为各层函数创建闭包时，只有在执行父函数（当前函数）的指令流时才会遇到为内层函数创建闭包的指令，那么父闭包自然要在子闭包之前创建。比如函数 0 中包含函数 1，函数 1 包含函数 2，创建函数 2 的闭包时，闭包 1 肯定已经创建过了，因此对于 718 行的 else 分支，upvalue 可以用索引 index 到当前函数闭包（所创建闭包的父闭包）中的 upvalues 数组中继承，反正用的也是父闭包的 upvalue，它指向哪我就跟着指哪。

接下来使 idx 自加 1 处理下一个 upvalue。

8. 第七部分操作码

这是指令执行中最后一部分，比较容易。

stepByStep/c7/d/vm/vm.c (8/8)

```
725
726         CASE(CREATE_CLASS): {
727             //指令流: 1 字节的 field 数量
728             //栈顶: 基类   次栈顶: 子类名
729
730             uint32_t fieldNum = READ_BYTE();
731             Value superClass = curThread->esp[-1];   //基类名
732             Value className = curThread->esp[-2];   //子类名
733
734             //回收基类所占的栈空间,
735             //次栈顶的空间暂时保留,创建的类会直接用该空间
736             DROP();
737
738             //校验基类合法性,若不合法则停止运行
739             validateSuperClass(vm, className, fieldNum, superClass);
740             Class* class = newClass(vm, VALUE_TO_OBJSTR(className),
741                     fieldNum, VALUE_TO_CLASS(superClass));
742
743             //类存储于栈底
744             stackStart[0] = OBJ_TO_VALUE(class);
745
746             LOOP();
```

```
747         }
748
749         CASE(INSTANCE_METHOD):
750         CASE(STATIC_METHOD): {
751             //指令流: 待绑定的方法“名字”在 vm->allMethodNames 中的 2 字节的索引
752             //栈顶: 待绑定的类  次栈顶: 待绑定的方法
753
754             //获得方法名的索引
755             uint32_t methodNameIndex = READ_SHORT();
756
757             //从栈顶中获得待绑定的类
758             Class* class = VALUE_TO_CLASS(PEEK());
759
760             //从次栈顶中获得待绑定的方法,
761             //这是由 OPCODE_CREATE_CLOSURE 操作码生成后压到栈中的
762             Value method = PEEK2();
763
764             bindMethodAndPatch(vm, opCode, methodNameIndex, class, method);
765
766             DROP();
767             DROP();
768             LOOP();
769         }
770
771         CASE(END):
772             NOT_REACHED();
773     }
774     NOT_REACHED();
775
776     #undef PUSH
777     #undef POP
778     #undef DROP
779     #undef PEEK
780     #undef PEEK2
781     #undef LOAD_CUR_FRAME
782     #undef STORE_CUR_FRAME
783     #undef READ_BYTE
784     #undef READ_SHORT
785 }
```

操作码 CREATE_CLASS 用于创建类，它需要 3 个参数：①指令流中是域的个数；②基类位于栈顶；③子类名位于次栈顶。第 730～732 行先获取这 3 个参数。紧接着调用 DROP 回收栈顶空间，此时之前的次栈顶便是栈顶了。接下来调用 validateSuperClass 校验基类合法性，然后通过函数 newClass 创建类 class，最后将 class 存储于运行时栈的栈底 stackStart[0]。说明一下，此时并未实际应用运行时栈（比如在栈中分配局部变量的空间或压入方法的实参），现在的栈底就是栈顶，故此时的 stackStart[0]就是之前用于存储子类名的次栈顶，也就是说，把 OBJ_TO_VALUE(class)存储到 curThread->esp[-1]也是正确的。

操作码 INSTANCE_METHOD 和 STATIC_METHOD 用于把方法绑定到类，这两个操作码都需要 3 个参数，待绑定的方法名的索引位于指令流中，待绑定的类位于栈顶，待绑定的方法体位

于次栈顶。第 754～762 行分别获取这些操作数。最后调用 bindMethodAndPatch 实现方法的绑定。后面的两个 DROP 是回收栈顶及次栈顶的空间。

7.5 运行虚拟机（本节源码 stepByStep/c7/e）

现在的虚拟机基本上已经完成了，至少是可以执行指令了，我们之前已经做了大量的编译方面的工作，经过编译后，只要打开相应的“扳机”，虚拟机便开始执行我们生成的指令。

首先要改的是 core.c 中的函数 executeModule，加上调用执行指令的函数 executeInstruction，如第 205 行所示。

stepByStep/c7/e/vm/core.c

```
…略
202 //执行模块
203 VMResult executeModule(VM* vm, Value moduleName, const char* moduleCode) {
204     ObjThread* objThread = loadModule(vm, moduleName, moduleCode);
205     return executeInstruction(vm, objThread);
206 }
…略
```

之前第 205 行只是简单的一句 return VM_RESULT_ERROR;。

在 compiler.c 中有两处修改，首先把之前编写好的编译函数添加到函数 compileProgram 中去，然后在函数 compileModule 中更新一部分代码，如下所示。

stepByStep/c7/e/compiler/compiler.c

```
…略
2333 //编译程序
2334 static void compileProgram(CompileUnit* cu) {
2335     if (matchToken(cu->curParser, TOKEN_CLASS)) {
2336         compileClassDefinition(cu);
2337     } else if (matchToken(cu->curParser, TOKEN_FUN)) {
2338         compileFunctionDefinition(cu);
2339     } else if (matchToken(cu->curParser, TOKEN_VAR)) {
2340         compileVarDefinition(cu, cu->curParser->preToken.type == TOKEN_STATIC);
2341     } else if (matchToken(cu->curParser, TOKEN_IMPORT)) {
2342         compileImport(cu);
2343     } else {
2344         compileStatment(cu);
2345     }
2346 }
2347
2348 //编译模块
2349 ObjFn* compileModule(VM* vm, ObjModule* objModule, const char* moduleCode) {
…略
2363     CompileUnit moduleCU;
2364     initCompileUnit(&parser, &moduleCU, NULL, false);
2365
2366     //记录现在模块变量的数量,后面检查预定义模块变量时可减少遍历
2367     uint32_t moduleVarNumBefor = objModule->moduleVarValue.count;
2368
```

```
2369     //初始的 parser->curToken.type 为 TOKEN_UNKNOWN,下面使其指向第一个合法的 token
2370     getNextToken(&parser);
2371
2372     //编译模块
2373     while (!matchToken(&parser, TOKEN_EOF)) {
2374         compileProgram(&moduleCU);
2375     }
2376
2377     //模块编译完成,生成 return null 返回,避免执行下面 endCompileUnit 中添加的 OPCODE_END
2378     writeOpCode(&moduleCU, OPCODE_PUSH_NULL);
2379     writeOpCode(&moduleCU, OPCODE_RETURN);
2380
2381     //检查在函数 id 中用行号声明的模块变量是否在引用之后有定义
2382     uint32_t idx = moduleVarNumBefor;
2383     while (idx < objModule->moduleVarValue.count) {
2384         //为简单起见,依然是遇到第一个错后就报错退出,后面的不再检查
2385         if (VALUE_IS_NUM(objModule->moduleVarValue.datas[idx])) {
2386             char* str = objModule->moduleVarName.datas[idx].str;
2387             ASSERT(str[objModule->moduleVarName.datas[idx].length] == '\0',
2388                    "module var name is not closed!");
2389             uint32_t lineNo = VALUE_TO_NUM(objModule->moduleVarValue.datas[idx]);
2390             COMPILE_ERROR(&parser, "line:%d, variable \'%s\' not defined!", lineNo, str);
2391         }
2392         idx++;
2393     }
2394
2395     //模块编译完成,当前编译单元置空
2396     vm->curParser->curCompileUnit = NULL;
2397     vm->curParser = vm->curParser->parent;
2398 #if DEBUG
2399     return endCompileUnit(&moduleCU, "(script)", 8);
2400 #else
2401     return endCompileUnit(&moduleCU);
2402 #endif
2403 }
```

之前在第 2377 行的代码是输出 There is something to do...后就退出，现在换上了正式的代码。第 2378～2379 行生成的两个操作码相当于“return null”，这是模块编译完成后给模块追回的代码，目的是可从模块中直接返回，因为后面会调用 endCompileUnit 给模块添加 OPCODE_END 伪操作码，如果没有此处的 return null，则虚拟机将执行到 OPCODE_END，这就出错了，OPCODE_END 只是程序结束的标记，属于伪操作码，永远不应该被执行到。接下来的 while 循环用于检查那些引用时尚未定义的模块变量是否在后面得到了定义，模块变量只允许提前引用，但后面必须有其定义，此处说的就是之前我们讨论好久的“伏笔”——用行号作为模块变量的值预先声明，留待编译后再检查，此时正是检查的时候。文件就是模块，现在模块编译完了，也就知道了那些预先声明的模块变量是否已经定义了。回忆一下，我们是在函数 id 中用行号作为变量值预先声明的模块变量，行号是数字，数字的类型是 VT_NUM，编译时变量的类型不会是 VT_NUM，只能是 VT_NULL 或 VT_OBJ 等，只有在运行时变量值的类型才有可能是 VT_NUM。如果有定义的话，我们就会用其实际定义的值的类型替换 VT_NUM，所以，此处发现变量值类型为 VT_NUM，说

明是漏定义的模块变量，应该报错。

从第 2380 行到最后就是一些收尾工作，不说了。

目前的编译器还不够完善，下一章还要继续完成另一部分的原生方法，目前我们只能编译处理核心代码 core.script.inc，因此我们只在编译此文件后就退出，不编译用户的脚本文件，这是在 cli.c 中临时修改的，代码如下。

stepByStep/c7/e/cli/cli.c

```
…略
 8 //执行脚本文件
 9 static void runFile(const char* path) {
10      const char* lastSlash = strrchr(path, '/');
11      if (lastSlash != NULL) {
12           char* root = (char*)malloc(lastSlash - path + 2);
13           memcpy(root, path, lastSlash - path + 1);
14           root[lastSlash - path + 1] = '\0';
15           rootDir = root;
16      }
17
18      VM* vm = newVM();
19      printf("There is something to do...\n"); exit(0);
20      const char* sourceCode = readFile(path);
21
22           executeModule(vm, OBJ_TO_VALUE(newObjString(vm, path, strlen(path))),
             sourceCode);
23 }
…略
```

临时代码加在了第 19 行，照样是输出 There is something to do...后就退出。下面编译运行，生成可执行文件 spr，运行效果如下所示。

```
[work@work a]$ ./spr /tmp/t.sp
There is something to do...
[work@work a]$
```

说明一下，目前 spr 的参数不起作用，我们还没到处理参数时就打印 There is something to do...后退出了，因此 spr 的参数可随便指定，所以并未列出/tmp/t.sp 中的代码。

第8章　内建类及其方法

大部分核心方法我们都是用 C 语言实现的，即原生方法，本章我们要继续补充原生方法。这类方法都是属于某个类，我们在 core.script.inc 中已经预先定义了这些类，有的只是空类，有的只是包含部分方法，总之它们并不完全，也正因为如此，我们才需要为其补全方法。

8.1 Bool 类及其方法（本节源码 stepByStep/c8/a）

布尔（Bool）类我们只简单地在 core.script.inc 中定义了空壳子，即 class Bool {}，下面在 core.c 中为其补充方法。

stepByStey/c8/a/vm/core.c

```
…略
 80 //返回核心模块 name 的 value 结构
 81 static Value getCoreClassValue(ObjModule* objModule, const char* name) {
 82     int index = getIndexFromSymbolTable(&objModule->moduleVarName, name, strlen(name));
 83     if (index == -1) {
 84         char id[MAX_ID_LEN] = {'\0'};
 85         memcpy(id, name, strlen(name));
 86         RUN_ERROR("something wrong occur: missing core class \"%s\"!", id);
 87     }
 88     return objModule->moduleVarValue.datas[index];
 89 }
…略
170 //返回 bool 的字符串形式: true 或 false
171 static bool primBoolToString(VM* vm, Value* args) {
172     ObjString* objString;
173     if (VALUE_TO_BOOL(args[0])) {  //若为 VT_TRUE
174         objString = newObjString(vm, "true", 4);
175     } else {
176         objString = newObjString(vm, "false", 5);
177     }
178     RET_OBJ(objString);
179 }
180
181 //bool 值取反
182 static bool primBoolNot(VM* vm UNUSED, Value* args) {
183     RET_BOOL(!VALUE_TO_BOOL(args[0]));
184 }
…略
```

```
305 //编译核心模块
306 void buildCore(VM* vm) {
…略
347       //执行核心模块
348       executeModule(vm, CORE_MODULE, coreModuleCode);
349
350       //Bool 类定义在 core.script.inc 中,将其挂载 Bool 类到 vm->boolClass
351        vm->boolClass = VALUE_TO_CLASS(getCoreClassValue(coreModule, "Bool"));
352       PRIM_METHOD_BIND(vm->boolClass, "toString", primBoolToString);
353       PRIM_METHOD_BIND(vm->boolClass, "!", primBoolNot);
354}
```

以下函数都是原生方法，后面会利用 PRIM_METHOD_BIND 将其与脚本中所用的方法名绑定。

函数 getCoreClassValue 用于在模块 objModule 中返回类 name 的 Value 结构。调用 getIndexFrom SymbolTable 从模块 objModule 中获取类名的索引，然后第 88 行根据此索引返回相应类的 Value 结构。

函数 primBoolToString 返回 bool 值的字符串形式，如 true 则返回 true，false 则返回 false，很简单不说了。

函数 primBoolNot 用于对 bool 值取反，如!true 则为 false，!false 则为 true。

以上的原生方法需要与脚本中的方法关联，比如脚本中若有代码!false，虚拟机会调用 primBoolNot 方法。实现关联的代码还是添加到 buildCore 中，第 350 行以下便是新增的部分。

脚本中的 Bool 类已经在函数 executeModule 中被编译过了，因此它已被注册到核心数据结构中，第 351 行将其通过 getCoreClassValue 读出并挂载到 vm->boolClass。

接下来调用 PRIM_METHOD_BIND 把脚本中的 toString 方法绑定到原生方法 primBoolToString，把脚本中的!绑定到 primBoolNot。

8.2 线程类及其方法（本节源码 stepByStep/c8/b）

实现多任务的目的是为了实现并发，调度器会轮换各个任务轮流上 CPU 运行，如果优先级相同的话，所有任务就拥有平等的运行机会。在调度器眼里，线程和进程一样都是可调度的任务，如果任务愿意的话，就可以主动调用 yield 方法让出 CPU 资源，还有一些情况下调度器会强迫任务下 CPU，比如阻塞的时候。

任何时候程序中都有一个主线程，简单来说就是 main 函数，它是程序的入口，它就相当于主线程。在脚本语言中初始运行的执行流就是主线程，后面创建了其他的线程后，程序中才有多个执行流，也就是说程序都是从主线程开始运行的，如果把主线程称为第一个线程，那么第二个线程肯定是由主线程调用的，其他线程可以由主线程或者另外的线程调用。

我们的线程并不是严格意义上的线程，因为我们在调度时并没有实现真正的轮换。线程一旦运行，要么直接运行完，要么由线程自己调用 yield 方法主动让出 CPU，这样新的线程才能运行，也就是说各线程享受 CPU 资源并不“公平”（希望以后有机会改进）。请看下面示例代码：

```
[work@work spr]$ cat -n /tmp/t2.sp
     1var t = Thread.new(Fn.new{
     2       System.print(111)
     3        var ret = Thread.yield()
     4        System.print(ret)
     5})
```

```
    6t.call()
    7System.print("wake t up")
    8t.call(222)
[work@work spr]$ ./spr /tmp/t2.sp
111
wake t up
222
[work@work spr]$
```

创建线程的方式是 Thread.new（线程中运行的函数）。第 1 行我们通过 Fn.new 的方式创建了函数，尽管此函数并没有名字，但 Thread.new 是通过 new 方法的形参来访问它的。第 2～4 行是函数体，函数体很简单，先打印 111，然后调用 yield 让出 CPU（此时回到主线程），最后再打印出 yield 的返回值 ret。线程创建后用变量 t 来存储，即 t 就是线程，我们的变量是弱类型。线程和函数的调用都是通过 call 方法，我们会在后面补充此方法。第 6 行在主线程中调用 t.call 运行线程 t，此时会打印出 111，随后调用 yield 让出 CPU，CPU 控制权回到主线程，此时第 7 行打印出 wake t up，第 8 行调用 t.call(222)，唤醒线程 t，也就是主线程让出 CPU 控制权给线程 t，参数 222 作为 yield 的返回值，第 3 行线程 t 恢复运行，变量 ret 的值便为 222，第 4 行打印出 222。

下面是我们线程的原生方法。

stepByStep/c8/b/vm/core.c

```
…略
 80 //校验是否是函数
 81 static bool validateFn(VM* vm, Value arg) {
 82       if (VALUE_TO_OBJCLOSURE(arg)) {
 83           return true;
 84       }
 85       vm->curThread->errorObj =
 86          OBJ_TO_VALUE(newObjString(vm, "argument must be a function!", 28));
 87       return false;
 88 }
…略
196 //以下以大写字符开头的为类名,表示类(静态)方法调用
197
198 //Thread.new(func):创建一个 thread 实例
199 static bool primThreadNew(VM* vm, Value* args) {
200       //代码块为参数必为闭包
201       if (!validateFn(vm, args[1])) {
202           return false;
203       }
204
205              ObjThread*objThread=newObjThread(vm,VALUE_TO_OBJCLOSURE(args[1]));
206
207       //使 stack[0]为接收者,保持栈平衡
208       objThread->stack[0] = VT_TO_VALUE(VT_NULL);
209       objThread->esp++;
210       RET_OBJ(objThread);
211 }
212
213 //Thread.abort(err):以错误信息 err 为参数退出线程
```

```
214 static bool primThreadAbort(VM* vm, Value* args) {
215     //此函数后续未处理,暂时放着
216     vm->curThread->errorObj = args[1]; //保存退出参数
217     return VALUE_IS_NULL(args[1]);
218 }
219
220 //Thread.current:返回当前的线程
221 static bool primThreadCurrent(VM* vm, Value* args UNUSED) {
222     RET_OBJ(vm->curThread);
223 }
224
225 //Thread.suspend():挂起线程,退出解析器
226 static bool primThreadSuspend(VM* vm, Value* args UNUSED) {
227     //目前 suspend 操作只会退出虚拟机,
228     //使 curThread 为 NULL,虚拟机将退出
229     vm->curThread = NULL;
230     return false;
231 }
232
233 //Thread.yield(arg)带参数让出 CPU
234 static bool primThreadYieldWithArg(VM* vm, Value* args) {
235     ObjThread* curThread = vm->curThread;
236     vm->curThread = curThread->caller;   //使 CPU 控制权回到主调方
237
238     curThread->caller = NULL;  //与调用者断开联系
239
240     if (vm->curThread != NULL) {
241         //如果当前线程有主调方,就将当前线程的返回值放在主调方的栈顶
242         vm->curThread->esp[-1] = args[1];
243
244         //对于 thread.yield(arg)来说, 回收 arg 的空间,
245         //保留 thread 参数所在的空间,将来唤醒时用于存储 yield 结果
246         curThread->esp--;
247     }
248     return false;
249 }
250
251 //Thread.yield() 无参数让出 CPU
252 static bool primThreadYieldWithoutArg(VM* vm, Value* args UNUSED) {
253     ObjThread* curThread = vm->curThread;
254     vm->curThread = curThread->caller;   //使 CPU 控制权回到主调方
255
256     curThread->caller = NULL;  //与调用者断开联系
257
258     if (vm->curThread != NULL) {
259         //为保持通用的栈结构,如果当前线程有主调方,
260         //就将空值作为返回值放在主调方的栈顶
261         vm->curThread->esp[-1] = VT_TO_VALUE(VT_NULL) ;
262     }
263     return false;
```

```
264 }
265
266 //切换到下一个线程 nextThread
267 static bool switchThread(VM* vm,
268         ObjThread* nextThread, Value* args, bool withArg) {
269     //在下一线程 nextThread 执行之前,其主调线程应该为空
270     if (nextThread->caller != NULL) {
271         RUN_ERROR("thread has been called!");
272     }
273     nextThread->caller = vm->curThread;
274
275     if (nextThread->usedFrameNum == 0) {
276         //只有已经运行完毕的 thread 的 usedFrameNum 才为 0
277         SET_ERROR_FALSE(vm, "a finished thread can't be switched to!");
278     }
279
280     if (!VALUE_IS_NULL(nextThread->errorObj)) {
281         //Thread.abort(arg)会设置 errorObj, 不能切换到 abort 的线程
282         SET_ERROR_FALSE(vm, "a aborted thread can't be switched to!");
283     }
284
285     //如果 call 有参数,回收参数的空间,
286     //只保留此栈顶用于存储 nextThread 返回后的结果
287     if (withArg) {
288         vm->curThread->esp--;
289     }
290
291     ASSERT(nextThread->esp > nextThread->stack, "esp should be greater than stack!");
292     //nextThread.call(arg)中的 arg 作为 nextThread.yield 的返回值
293     //存储到 nextThread 的栈顶,否则压入 null 保持栈平衡
294     nextThread->esp[-1] = withArg ? args[1] : VT_TO_VALUE(VT_NULL);
295
296     //使当前线程指向 nextThread,使之就绪
297     vm->curThread = nextThread;
298
299     //返回 false 以进入 vm 中的切换线程流程
300     return false;
301 }
302
303 //objThread.call()
304 static bool primThreadCallWithoutArg(VM* vm, Value* args) {
305     return switchThread(vm, VALUE_TO_OBJTHREAD(args[0]), args, false);
306 }
307
308 //objThread.call(arg)
309 static bool primThreadCallWithArg(VM* vm, Value* args) {
310     return switchThread(vm, VALUE_TO_OBJTHREAD(args[0]), args, true);
311 }
312
313 //objThread.isDone 返回线程是否运行完成
```

```
314 static bool primThreadIsDone(VM* vm UNUSED, Value* args) {
315       //获取.isDone 的调用者
316       ObjThread* objThread = VALUE_TO_OBJTHREAD(args[0]);
317             RET_BOOL(objThread->usedFrameNum == 0 || !VALUE_IS_NULL(objThread->errorObj));
318 }
…略
489       //Thread 类也是在 core.script.inc 中定义的,
490       //将其挂载到 vm->threadClass 并补充原生方法
491       vm->threadClass = VALUE_TO_CLASS(getCoreClassValue(coreModule, "Thread"));
492       //以下是类方法
493       PRIM_METHOD_BIND(vm->threadClass->objHeader.class, "new(_)", primThreadNew);
494       PRIM_METHOD_BIND(vm->threadClass->objHeader.class, "abort(_)", primThreadAbort);
495       PRIM_METHOD_BIND(vm->threadClass->objHeader.class, "current", primThreadCurrent);
496       PRIM_METHOD_BIND(vm->threadClass->objHeader.class, "suspend()", primThreadSuspend);
497       PRIM_METHOD_BIND(vm->threadClass->objHeader.class, "yield(_)",
           primThreadYieldWithArg);
498       PRIM_METHOD_BIND(vm->threadClass->objHeader.class, "yield()",
          primThreadYieldWithoutArg);
499       //以下是实例方法
500       PRIM_METHOD_BIND(vm->threadClass, "call()", primThreadCallWithoutArg);
501       PRIM_METHOD_BIND(vm->threadClass, "call(_)", primThreadCallWithArg);
502       PRIM_METHOD_BIND(vm->threadClass, "isDone", primThreadIsDone);
503 }
```

函数 validateFn 用于校验参数 arg 是否为函数。如果可以通过宏 VALUE_TO_OBJCLOSURE 转换为闭包，就返回 true。否则把报错信息放在线程的 errorObj 成员中，随后返回 false。

继续介绍之前先说明一下，类（静态）方法是以类名来调用的，类名是以大写字符开头（通常实例对象都以小写开头，但不限），在以下各函数的功能注释中，若以大写字符开头的符号则为类名。而且在我们语言中，类方法不能通过实例对象调用，即使在实例的代码中，为了使用类方法，必须严格以“类名.方法”的形式调用。

原生方法的参数存储在数组 args 中，我们在虚拟机的 CALL0～CALL16 和 SUPER0～SUPER16 指令族中使指针 args 指向运行时栈中的参数，这样就为 args 填充了参数，其中 args[0]是消息的接收者，即实例对象或类，args[1]是原生方法的第 1 个参数，其他依次类推。

函数 primThreadNew 用于创建一个 thread 实例，它是 Thread 类中的静态方法 new 对应的原生实现，即脚本中执行 Thread.new(func)时我们会调用 primThreadNew，脚本中的参数 func 是线程中待执行的函数，func 对应于 primThreadNew 的形参 args[1]。必须确保 args[1]是函数才能加入到线程中运行，因此先调用 validateFn 校验 args[1]是否为合法的函数，如果是就调用 newObjThread 创建线程。函数和方法的运行离不开堆栈框架，堆栈框架中最重要的就是运行时栈了，虚拟机会在方法或函数所属线程的“大栈”中分一块下来作为它们的运行时栈。在函数 newObjThread 中会调用 prepareFrame 为所运行的闭包（函数或方法）创建堆栈框架，这其中包括准备运行时栈的工作，第一个在线程中运行的闭包，其运行时栈 stackStart[0]正是线程“大栈”的 stack[0]，我们又会把各闭包运行时栈的 stackStart[0]用于存储消息的接收者，即对象实例或类，故此时用 VT_TO_VALUE 把 stack[0]（即线程中第一个闭包的 stackStart[0]）初始化为 NULL 来占位，然后使 esp 自加 1 更新栈顶以空出来 stack[0]，最后返回创建的线程实例 objThread。

函数 primThreadAbort 用于以错误信息 args[1]为参数退出线程，它是脚本中调用 Thread.abort(err)

时执行的原生方法，其中 err 便是 args[1]。此函数目前处理比较简单，就是把 args[1]放在线程的 errorObj 中。方法是由虚拟机调用的，原生方法是在 vm.c 中 invokeMethod 标号后的 case MT_PRIMITIVE 中处理，理想情况下从原生方法返回后虚拟机应该判断 errorObj 是否为空，如果非空说明原生方法出现了异常情况，然后对此做进一步的处理，比如异常捕获，不过目前虚拟机中尚未完成此功能，以后有机会再说吧。

函数 primThreadCurrent 用于返回当前的线程，脚本中调用 Thread.current 时就是调用本函数，实现很简单，就是返回 vm->curThread。

函数 primThreadSuspend 用于挂起线程，脚本中调用 Thread.suspend()时就执行本函数，使 vm->curThread 为 NULL 就行了，虚拟机检测到 curThread 为 NULL 时表示目前没有可运行的线程了，会退出解析器，以后也没有机会再 resume 了。

函数 primThreadYieldWithArg 用于带参数（args[1]）让出 CPU 使用权，脚本中调用 Thread.yield(arg)就是调用本函数。无论是否带参数，执行 yield 方法就是让出 CPU，我们在本小节开头举的例子就是 yield 方法调用，不过那是无参数 yield，下面举个 yield 带参数的例子。

```
[work@work spr]$ cat -n /tmp/newt2.sp
     1var t = Thread.new(Fn.new{
     2    System.print("t: I am running")
     3    var ret = Thread.yield("see you again")
     4    System.print("main say: " + ret.toString)
     5})
     6
     7var tRet = t.call()
     8System.print("t say: " + tRet)
     9System.print("main: wake t up")
    10t.call("welcome back")
[work@work spr]$ ./spr /tmp/newt2.sp
t: I am running
t say: see you again
main: wake t up
main say: welcome back
[work@work spr]$
```

大家看看这个例子就明白 yield(arg)的用法了。

yield（让出）CPU 就是切换线程到主调方。虚拟机始终以 vm->curThread 指向的进程为就绪或当前线程，因此函数前 3 行使 vm->curThread 指向当前线程的主调方，目的是使虚拟机在本原生方法执行后可切换到当前线程的主调方，即交回 CPU 控制权给调用当前线程的那个线程。第 238 行使当前线程的主调方为 NULL，断开了与主调方的调用关系。第 240 行判断若主调方不为空，也就是当前线程有主调方的话（主线程没有主调方，主线程相当于程序运行时的第一个执行流，比如 main 函数），就在第 242 行将当前线程的结果即参数 args[1]放在主调方栈顶，目的是使主调方可获得当前线程执行 yield 时的参数（如上面脚本例子中的 see you again）。注意，此时说的是往主调方栈中存储当前线程调用 yield 方法的参数，也就是当前线程要让出 CPU 了，希望给主调方传递参数。下面是当前线程中为存储主调方传给本（当前）线程的参数所做的准备工作，此时尚未存储，只是准备工作，也就是指将来本（当前）线程恢复运行时（由主调方通过该线程实例对象的 call 方法使该线程激活运行，注意，call 方法是实例方法，所以可以判断激活哪个线程对象），我们将存储 call 的参数在待运行线程（当前线程）的栈顶，这样待运行的线程便可拿到主调方提供的参数（比如上面脚本中的 welcome back）。此时是当前线程在调用本原生函数，对于

脚本来说，当前方法调用 Thread.yield(arg)拥有两个参数即 Thread 和 arg，那么这两个参数占用了栈中两个 slot，由于存储主调方传递的参数只需要一个 slot，故丢弃栈顶空间，预留之前的次栈顶空间就够用了。即丢弃 Thread.yield(arg)中的 arg，也就是 args[1]，只保留 args[0]用于存储主调方线程通过 call 方法传递的参数，因此 esp 自减 1 就行。最后返回 false 给虚拟机，表示切换线程，我们都是通过返回 false 来告诉虚拟机需要切换线程的。如果有主调方的话，此时虚拟机就会切换到主调方线程，因为虚拟机是从 vm->curThread 获取当前运行的线程，我们在函数开头已经更新它为当前线程的主调方了。如果 curThread 为 NULL，虚拟机就认为全部线程执行完毕，可以退出了。本函数可以和后面的 primThreadCallWithArg 函数对照着看。

函数 primThreadYieldWithoutArg 用于无参数让出 CPU，相当于脚本中执行 Thread.yield()，思路和函数 primThreadYieldWithArg 类似，只是少了给主调方线程返回参数的步骤。

函数 switchThread 用于切换到下一个线程 nextThread，其中参数 withArg 表示是否带参数调用。脚本中并不能直接用到此函数，这是给脚本中的 objThread.call()和 objThread.call(arg)准备的。本函数开头先检测几种不能切换的情况。线程对象的运行是通过调用该线程对象的 call 方法实现的，也就是说 call 方法是被调用的，因此肯定有个主调方。下一个待运行的线程 nextThread 在运行之前，主调方肯定为空，函数开头便通过 nextThread->caller 是否为 NULL 来判断这种情况，如果不为 NULL 则报错。任意时刻当前运行的线程都记录在 vm->curThread，任何待运行的线程肯定都是由当前运行的线程调用的，因此接下来使下一线程的主调方即 nextThread->caller 指向当前线程 vm->curThread。线程是用来运行闭包（函数或方法）的容器，执行闭包就是线程的任务，闭包的运行是需要堆栈框架的，如果线程中已使用的堆栈框架数量为 0，即 ObjThread->usedFrameNum 为 0，则说明线程已经完成了使命，属于已经执行完毕的线程，而这种没有执行任务的线程是不应该再调用的，对于这种情况会在第 277 行报错。如果线程在上次运行时已经出错了，其 errorObj 成员中就会记录出错的对象，即出错的线程，其 errorObj 成员肯定不为 NULL。已经出错的线程也是不允许调用的，对于这种情况会在第 282 行报错。大家要清楚，方法调用之前肯定要为方法准备参数，参数是存储在主调方自己的运行时栈中，对应到线程运行也是一样的，尽管待运行线程 nextThread 能运行是调用了该线程对象的 call 方法，但是 call 方法的参数可是存储在当前线程 curThread 的栈中，确切地说是存储在 curThread 中当前正在运行的闭包的运行时栈中，毕竟一个函数中可以嵌套运行多个闭包。当前线程 curThread 正要交出 CPU 使用权给 nextThread，待将来 nextThread 运行完毕后，CPU 控制权要交回给主调方即 curThread，那时 curThread 则恢复运行，它需要从栈顶获取此时所调用的 nextThread 线程的返回值，此时我们需要把 nextThread 的返回值存储到主调方即当前线程 curThread 的栈顶，这样 curThread 才能拿到返回值。不过此时涉及栈空间是否合适的问题，后面会看到 call 方法有两种，一种不带参数即 call()，另一种带参数即 call(arg)。下面分情况讨论，如果主调方是通过 nextThread.call()无参数的形式激活 nextThread 运行，那么主调方栈中只有一个参数即 nextThread，此时正好用 nextThread 在栈中的 slot 来存储将来 nextThread 返回后的返回值。如果主调方是通过 nextThread.call(arg)有参数的方式调用 nextThread，那么 withArg 为 true，主调线程 curThread 的栈中分别是 nextThread 和 arg 占用的两个 slot，由于当前主调线程就只需要一个 slot 来存储 nextThread 的返回值，所以在第 288 行使 esp--，回收一个空间。提醒，这里修改的是主调方线程 curThread 的栈，不是 nextThread（被调方）的栈。接下来把 call 方法的参数 arg 作为执行过 yield 方法的线程的返回值，也就是说，线程 nextThread 之前若调用 yield 让出了 CPU 控制权给主调方 curThread，这次主调方又通过 nextThread.call(arg)使 nextThread 恢复运行，那么 arg 将成为 nextThread.yield()的返回值，这里的 arg 就是指上面脚本例子中的 welcome back。第 294 行根据 withArg 的值往 nextThread 的栈顶装入参数或空值，空值是为了保持栈平衡。虚拟机是以 vm->curThread 来确定当前运行的线程，因此

使 vm->curThread 指向 nextThread 使之就绪，最后返回 false 使虚拟机进行线程切换。

接下来的函数 primThreadCallWithoutArg 和 primThreadCallWithArg 分别表示脚本中的 objThread.call()和 objThread.call(arg)，即无参调用和有参调用（只有一个参数），这两个函数都是对 switchThread 的封装，注意，这两个函数都是实例方法，可不是前面介绍的类方法。

函数 primThreadIsDone 用于返回线程是否运行完成，对于脚本中的 objThread.isDone，同样也是实例方法。参数数组 args 中的 args[0]元素始终是消息的接收者，这里是实例对象，用 objThread 指针变量指向其转换后的实例线程对象。线程是否运行完毕，一是体现在堆栈框架的使用量，即 usedFrameNum 是否为 0，二是该线程是否有错误出现，即 errorObj 是否有值。满足这两种情况之一便认为线程运行结束。

最后第 489～502 行是把上面实现的原生方法与脚本方法绑定，之前介绍原生方法关联时已经说过，不赘述了，不过要注意的是，线程实例方法是注册到 vm->threadClass 中，即线程对象所属类，类方法是注册到 threadClass 类的 meta 类，即 vm->threadClass->objHeader.class。

8.3 函数类及其方法和函数调用重载（本节源码 stepByStep/c8/c）

本节完成函数类及函数调用的原生方法。函数调用最大支持 16 个参数，这涉及函数重载。首先我们要分清楚方法调用和函数调用的区别，如下所示。

```
[work@work spr]$ cat -n /tmp/test.sp
     1 class test {
     2     new() {
     3     }
     4
     5     method(para){
     6        System.print(para)
     7     }
     8 }
     9 var t = test.new()
    10 t.method("method call")
    11
    12 t = Fn.new{|para|
    13     System.print(para)
    14 }
    15 t.call("function call")
    16
    17 fun tt(para) {
    18     System.print(para)
    19 }
    20  tt("function call normal form")
[work@work spr]$ ./spr /tmp/test.sp
method call
function call
function call normal form
[work@work spr]$
```

第 1～10 行定义一个类 test，包含名为 method 的方法，第 10 行传入参数 method call 调用方

法 method。

第 12～15 行通过 Fn.new()方法生成一个函数，也许你觉得我写错了，明明是 Fn.new，后面哪来的小括号？是的，后面没有小括号，可是后面有大括号，整个大括号就是块参数，其中包含的就是所创建函数的函数体，之前在编译时说过，块参数也是方法的一个参数，在此它作为 new 方法的 args[1]，因此 new 方法的签名是 new(_)。大家要是想看 Fn 类的实现，还是要看文件 core.script.inc，不过那也是空的，只是提前占个名而已，留待本节往此类中装入方法。通常块参数都是函数体，相当于匿名函数，是函数就要有参数，以块参数实现的函数，其参数是用“|paralist|”实现的，这和很多其他语言采取的方式是一样的，比如 ruby。函数是用 call 方法来调用的，本节就要实现这个方法。块参数就是个函数体，new 方法就是直接返回块参数即 new 方法的参数 args[1]，因此 t 就是函数闭包，所以参数 function call 就直接传给了形参 para。

第 17～20 行定义的函数 tt 是函数 t 的等价形式。

本节要完成第 12～15 行的部分。

stepByStep/c8/c/vm/core.c

```
…略
320 //Fn.new(_):新建一个函数对象
321 static bool primFnNew(VM* vm, Value* args) {
322       //代码块为参数必为闭包
323       if (!validateFn(vm, args[1])) return false;
324
325       //直接返回函数闭包
326       RET_VALUE(args[1]);
327 }
…略
448 //绑定 fn.call 的重载
449 static void bindFnOverloadCall(VM* vm, const char* sign) {
450       uint32_t index = ensureSymbolExist(vm, &vm->allMethodNames, sign, strlen(sign));
451       //构造 method
452       Method method = {MT_FN_CALL, {0}};
453       bindMethod(vm, vm->fnClass, index, method);
454 }
…略
521       //绑定函数类
522       vm->fnClass = VALUE_TO_CLASS(getCoreClassValue(coreModule, "Fn"));
523             PRIM_METHOD_BIND(vm->fnClass->objHeader.class, "new(_)", primFnNew);
524
525       //绑定 call 的重载方法
526       bindFnOverloadCall(vm, "call()");
527       bindFnOverloadCall(vm, "call(_)");
528       bindFnOverloadCall(vm, "call(_,_)");
529       bindFnOverloadCall(vm, "call(_,_,_)");
530       bindFnOverloadCall(vm, "call(_,_,_,_)");
531       bindFnOverloadCall(vm, "call(_,_,_,_,_)");
532       bindFnOverloadCall(vm, "call(_,_,_,_,_,_)");
533       bindFnOverloadCall(vm, "call(_,_,_,_,_,_,_)");
534       bindFnOverloadCall(vm, "call(_,_,_,_,_,_,_,_)");
535       bindFnOverloadCall(vm, "call(_,_,_,_,_,_,_,_,_)");
536       bindFnOverloadCall(vm, "call(_,_,_,_,_,_,_,_,_,_)");
```

```
537     bindFnOverloadCall(vm, "call(_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_)");
538     bindFnOverloadCall(vm, "call(_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_)");
539     bindFnOverloadCall(vm, "call(_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_)");
540     bindFnOverloadCall(vm, "call(_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_)");
541     bindFnOverloadCall(vm, "call(_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_)");
542     bindFnOverloadCall(vm, "call(_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_)");
543 }
```

函数 primFnNew 就是脚本中的 Fn.new(_)，用于创建函数对象。前面说过，块参数即 args[1]是函数体，函数体就是闭包，因此先要确认 args[1]为闭包，如果是就直接返回函数体即 args[1]。

函数 bindFnOverloadCall 用于绑定 call 方法的重载。我们已经知道 call 的参数是 0～16，因此用同样的一个 call 方法支持 0～16 个参数，这就是对 call 方法的重载。参数 sign 的类型是 char*，并不是 Sign 结构，因此传进来的只是方法名字符串。调用 ensureSymbolExist 获取方法名的索引，然后封装 Method 结构，注意，方法类型是 MT_FN_CALL，这是针对函数的方法，可不是 MT_SCRIPT（类中定义的方法）。在 vm.c 中的对此类方法的处理可见 case MT_FN_CALL:中的 VALUE_TO_OBJCLOSURE(args[0])，这是把实例对象即消息接收者当成函数闭包，也就是说 args[0]是待调用的函数。最后调用 bindMethod 将方法与类 vm->fnClass 绑定。

第 522～523 行是获得类 Fn 到 vm->fnClass，然后把类方法 new()绑定到 fnClass 的 meta 类，因为 new 方法是静态方法。

最后调用 bindFnOverloadCall 依次把 call 的 17 种重载形式绑定。

8.4 Null 类及其方法（本节源码 stepByStep/c8/d）

本小节内容很少，往 Null 类中装入两个方法。还是先看下样例代码以便于大家理解。

```
[work@work spr]$ cat -n /tmp/test.sp
     1System.print(null)
     2System.print(!null)
     3System.print(null.toString)
[work@work spr]$ ./spr /tmp/test.sp
null
true
null
[work@work spr]$
```

第 1 行输出 null，结果就是 null，这很正常。第 2 行输出 null 的取非，结果为 true，第 3 行输出 null 的字符串形式，结果也是 null，不过这是字符串。尽管屏幕上输出的任何内容都是转换为字符串后输出的，但我们要清楚第 1 行的 null 原本是一种类型为 null 的数据，第 3 行的 null 原本就是字符串 null。

下面看实现代码。

stepByStep/c8/d/vm/core.c

```
...略
329 //null 取非
330 static bool primNullNot(VM* vm UNUSED, Value* args UNUSED) {
331     RET_VALUE(BOOL_TO_VALUE(true));
332 }
333
```

```
334 //null 的字符串化
335 static bool primNullToString(VM* vm, Value* args UNUSED) {
336       ObjString* objString = newObjString(vm, "null", 4);
337       RET_OBJ(objString);
338 }
…略
555       //绑定 Null 类的方法
556       vm->nullClass = VALUE_TO_CLASS(getCoreClassValue(coreModule, "Null"));
557       PRIM_METHOD_BIND(vm->nullClass, "!", primNullNot);
558       PRIM_METHOD_BIND(vm->nullClass, "toString", primNullToString);
559 }
```

8.5 Num 类及其方法（本节源码 stepByStep/c8/e）

Num 类就是数字类，在我们语言中数字也完全面向对象，先看下样例代码。

```
[work@work spr]$ cat -n /tmp/test.sp
     1System.print(1+2)
     2System.print(3.3.ceil)
     3System.print(3.3.floor)
[work@work spr]$ ./spr /tmp/test.sp
3
4
3
[work@work spr]$
```

第 1 行输出 1+2 的和，第 2 行输出 3.3 向上取整的结果，第 3 行输出 3.3 向下取整的结果。尽管都是数字表达式，但我们在处理上还采取了两种不同的编译方案，理由如下：

sparrow 是完全面向对象的语言（甚至函数都是对象），我们知道，运算符可以是类的方法，在此加号是数字类 Num 中的方法，上述样例中第 1 行的 1+2，表面看上去并不是面向对象的方式，如果严格按照面向对象的方式（对象.方法（参数））计算 1+2，用户所写的表达式就应该为 1.+(2)，显然，这太滑稽和刻意了。我们对此的处理方法是按照运算符和方法名来采取两种不同的编译策略。以中缀运算符+为例，把 1+2 按照 1.+(2)的逻辑来编译，让中缀运算符的编译方式执行 Rules 数组中的 infixOperator 函数，在此函数中将中缀运算符伪装成数字类的方法调用，有兴趣的读者可以翻回去复习一下 infixOperator 函数，在此函数中调用 emitCallBySignature 把运算符包装成方法签名（主要起作用的是 emitCallBySignature 中所调用的函数 sign2string），并把签名装入 vm->allMethodNames，最后生成调用该方法签名的 OPCODE_CALLx 操作码。其他中缀运算符采取相同的处理方法。而对于那些一般意义的显式的方法调用如上述的 3.3.ceil，编译时就遵循 Rules 数组中 TOKEN_DOT 对应的规则。

本节工作量有点大。

stepByStep/c8/e/vm/core.c (1/3)

```
…略
 86 //将数字转换为字符串
 87 static ObjString* num2str(VM* vm, double num) {
 88       //nan 不是一个确定的值，因此 nan 和 nan 是不相等的
 89       if (num != num) {
 90             return newObjString(vm, "nan", 3);
```

```
91      }
92
93      if (num == INFINITY) {
94          return newObjString(vm, "infinity", 8);
95      }
96
97      if (num == -INFINITY) {
98          return newObjString(vm, "-infinity", 9);
99      }
100
101     //以下 24 字节的缓冲区足以容纳双精度到字符串的转换
102     char buf[24] = {'\0'};
103     int len = sprintf(buf, "%.14g", num);
104     return newObjString(vm, buf, len);
105 }
106
107 //判断 arg 是否为数字
108 static bool validateNum(VM* vm, Value arg) {
109     if (VALUE_IS_NUM(arg)) {
110         return true;
111     }
112     SET_ERROR_FALSE(vm, "argument must be number!");
113 }
114
115 //判断 arg 是否为字符串
116 static bool validateString(VM* vm, Value arg) {
117     if (VALUE_IS_OBJSTR(arg)) {
118         return true;
119     }
120     SET_ERROR_FALSE(vm, "argument must be string!");
121 }
…略
383 //将字符串转换为数字
384 static bool primNumFromString(VM* vm, Value* args) {
385     if (!validateString(vm, args[1])) {
386         return false;
387     }
388
389     ObjString* objString = VALUE_TO_OBJSTR(args[1]);
390
391     //空字符串返回 RETURN_NULL
392     if (objString->value.length == 0) {
393         RET_NULL;
394     }
395
396         ASSERT(objString->value.start[objString->value.length] == '\0', "objString
don't teminate!");
397
398     errno = 0;
399     char* endPtr;
```

```
400
401       //将字符串转换为 double 型，它会自动跳过前面的空白
402       double num = strtod(objString->value.start, &endPtr);
403
404       //以 endPtr 是否等于 start+length 来判断不能转换的字符之后是否全是空白
405       while (*endPtr != '\0' && isspace((unsigned char)*endPtr)) {
406          endPtr++;
407       }
408
409       if (errno == ERANGE) {
410            RUN_ERROR("string too large!");
411       }
412
413       //如果字符串中不能转换的字符不全是空白，字符串非法，返回 NULL
414       if (endPtr < objString->value.start + objString->value.length) {
415            RET_NULL;
416       }
417
418       //至此，检查通过，返回正确结果
419       RET_NUM(num);
420 }
421
```

函数 num2str 用于把数字 num 转换为字符串对象，转换之前要确认 num 的类型。如果 num 的类型为 NaN，即 not a number，则表示非数字类型，NaN 的特点是数值不确定，与任何数字都不相等，包括它自身，因此函数开头利用此特性判断 num 是否为 NaN。如果是 NaN，调用 newObjString 把字符串 nan 生成字符串对象，并返回其指针。宏 INFINITY 表示正无穷大，如果 num 为正无穷大就生成字符串值为 infinity 的字符串对象并返回其指针。同理，如果 num 为负无穷大，就生成字符串值为-infinity 的字符串对象，并返回其指针。如果并不是以上 3 种类型，就在第 102～104 行利用 sprintf 函数把数字 num 转为字符串存储到 buf，然后生成字符串对象返回其指针。这里说明一下为什么 buf 的大小为 24 字节，本函数是把数字转换为字符串，我们这里以最长的双精度数字-1.12345678901234e-1022 为例，其中每个数字被一一转换为相应的字符，也就是一个字符是 1 字节大小，统计一下有多少个数字就可以了。我们从左到右统计，符号-占 1 字节，数字 1 占 1 字节，点“.”占 1 字节，接下来是 14 个十进制数字占 14 字节，e 占 1 字节，-（或+）占 1 字节，然后是 4 位指数部分占 4 字节，转换后的字符串的结束符\0 要占 1 字节，总共是 24 字节，因此 24 字节的缓冲区足以容纳双精度到字符串的转换。

接下来的函数 validateNum 和 validateString 分别用于校验 arg 是否为数字和字符串，是就返回 true，否则就报错并停止运行。

函数 primNumFromString 用于把字符串 args[1]转换为数字。在函数的开头校验 args[1]为合法的字符串后，将其转换为字符串对象并用 objString 指针指向此对象。本函数的功能主要是用函数 strtod 完成的，它的声明是 double strtod (const char* str, char** endptr)，其中 str 是要转换的字符串，若 endptr 不为 NULL，strtod 会把下一个待转换字符的地址放到 endptr 中，因此转换失败时，endptr 为第一个不能转换的字符的指针。strtod 函数会扫描参数 str 字符串，跳过前面的空白字符，例如空格、tab 缩进等，直到遇上数字或正负符号才开始做转换，直到出现非数字或字符串结束时(\0)才结束转换，并将结果返回。参数 str 可包含正负号、小数点或指数部分（用 E(e)表示），如 123.456 或 123e-2。strtod 是不管字符串中后面的空白字符的，即在转换过程中遇到了非数字字符（包括



空白）就退出。此时为了健壮性考虑，这里要判断后面不能转换的字符是否是空白，如果到结尾全是空白就认为合法，比如“1234”这种字符串还是可以解析出数字 1234 的，这是可接受的，这就是第 405～407 行和第 414 行代码的意图。如果待转换的字符串过长，strtod 函数会设置 error 变量的值为 ERANGE。执行到第 418 行时，说明字符串检查通过，返回转换后的数字。

stepByStep/c8/e/vm/core.c (2/3)

```
422 //返回圆周率
423 static bool primNumPi(VM* vm UNUSED, Value* args UNUSED) {
424     RET_NUM(3.14159265358979323846);
425 }
426
427 #define PRIM_NUM_INFIX(name, operator, type) \
428   static bool name(VM* vm, Value* args) {\
429     if (!validateNum(vm, args[1])) {\
430       return false; \
431     }\
432       RET_##type(VALUE_TO_NUM(args[0]) operator VALUE_TO_NUM(args[1]));\
433   }
434
435 PRIM_NUM_INFIX(primNumPlus,   +, NUM);
436 PRIM_NUM_INFIX(primNumMinus,  -, NUM);
437 PRIM_NUM_INFIX(primNumMul,    *, NUM);
438 PRIM_NUM_INFIX(primNumDiv,    /, NUM);
439 PRIM_NUM_INFIX(primNumGt,    >, BOOL);
440 PRIM_NUM_INFIX(primNumGe,   >=, BOOL);
441 PRIM_NUM_INFIX(primNumLt,    <, BOOL);
442 PRIM_NUM_INFIX(primNumLe,   <=, BOOL);
443 #undef PRIM_NUM_INFIX
444
445 #define PRIM_NUM_BIT(name, operator) \
446   static bool name(VM* vm UNUSED, Value* args) {\
447     if (!validateNum(vm, args[1])) {\
448       return false;\
449     }\
450     uint32_t leftOperand = VALUE_TO_NUM(args[0]); \
451     uint32_t rightOperand = VALUE_TO_NUM(args[1]); \
452     RET_NUM(leftOperand operator rightOperand);\
453   }
454
455 PRIM_NUM_BIT(primNumBitAnd, &);
456 PRIM_NUM_BIT(primNumBitOr, |);
457 PRIM_NUM_BIT(primNumBitShiftRight, >>);
458 PRIM_NUM_BIT(primNumBitShiftLeft, <<);
459 #undef PRIM_NUM_BIT
460
461 //使用数学库函数
462 #define PRIM_NUM_MATH_FN(name, mathFn) \
463   static bool name(VM* vm UNUSED, Value* args) {\
464     RET_NUM(mathFn(VALUE_TO_NUM(args[0]))); \
```




```
465     }
466
467 PRIM_NUM_MATH_FN(primNumAbs,      fabs);
468 PRIM_NUM_MATH_FN(primNumAcos,     acos);
469 PRIM_NUM_MATH_FN(primNumAsin,     asin);
470 PRIM_NUM_MATH_FN(primNumAtan,     atan);
471 PRIM_NUM_MATH_FN(primNumCeil,     ceil);
472 PRIM_NUM_MATH_FN(primNumCos,      cos);
473 PRIM_NUM_MATH_FN(primNumFloor,   floor);
474 PRIM_NUM_MATH_FN(primNumNegate,  -);
475 PRIM_NUM_MATH_FN(primNumSin,      sin);
476 PRIM_NUM_MATH_FN(primNumSqrt,     sqrt);  //开方
477 PRIM_NUM_MATH_FN(primNumTan,      tan);
478 #undef PRIM_NUM_MATH_FN
479
480 //这里用 fmod 实现浮点取模
481 static bool primNumMod(VM* vm UNUSED, Value* args) {
482     if (!validateNum(vm, args[1])) {
483         return false;
484     }
485     RET_NUM(fmod(VALUE_TO_NUM(args[0]), VALUE_TO_NUM(args[1])));
486 }
487
488 //数字取反
489 static bool primNumBitNot(VM* vm UNUSED, Value* args) {
490     RET_NUM(~(uint32_t)VALUE_TO_NUM(args[0]));
491 }
492
493 //[数字 from..数字 to]
494 static bool primNumRange(VM* vm UNUSED, Value* args) {
495     if (!validateNum(vm, args[1])) {
496         return false;
497     }
498
499     double from = VALUE_TO_NUM(args[0]);
500     double to = VALUE_TO_NUM(args[1]);
501     RET_OBJ(newObjRange(vm, from, to));
502 }
503
504 //atan2(args[1])
505 static bool primNumAtan2(VM* vm UNUSED, Value* args) {
506     if (!validateNum(vm, args[1])) {
507         return false;
508     }
509
510     RET_NUM(atan2(VALUE_TO_NUM(args[0]), VALUE_TO_NUM(args[1])));
511 }
512
513 //返回小数部分
514 static bool primNumFraction(VM* vm UNUSED, Value* args) {
```




```
515     double dummyInteger;
516     RET_NUM(modf(VALUE_TO_NUM(args[0]), &dummyInteger));
517 }
518
519 //判断数字是否无穷大,不区分正负无穷大
520 static bool primNumIsInfinity(VM* vm UNUSED, Value* args) {
521     RET_BOOL(isinf(VALUE_TO_NUM(args[0])));
522 }
523
524 //判断是否为数字
525 static bool primNumIsInteger(VM* vm UNUSED, Value* args) {
526     double num = VALUE_TO_NUM(args[0]);
527     //如果是 nan(不是一个数字)或无限大的数字就返回 false
528     if (isnan(num) || isinf(num)) {
529         RET_FALSE;
530     }
531     RET_BOOL(trunc(num) == num);
532 }
533
534 //判断数字是否为 nan
535 static bool primNumIsNan(VM* vm UNUSED, Value* args) {
536     RET_BOOL(isnan(VALUE_TO_NUM(args[0])));
537 }
538
539 //数字转换为字符串
540 static bool primNumToString(VM* vm UNUSED, Value* args) {
541     RET_OBJ(num2str(vm, VALUE_TO_NUM(args[0])));
542 }
543
544 //取数字的整数部分
545 static bool primNumTruncate(VM* vm UNUSED, Value* args) {
546     double integer;
547     modf(VALUE_TO_NUM(args[0]), &integer);
548     RET_NUM(integer);
549 }
550
551 //判断两个数字是否相等
552 static bool primNumEqual(VM* vm UNUSED, Value* args) {
553     if (!validateNum(vm, args[1])) {
554         RET_FALSE;
555     }
556
557     RET_BOOL(VALUE_TO_NUM(args[0]) == VALUE_TO_NUM(args[1]));
558 }
559
560 //判断两个数字是否不等
561 static bool primNumNotEqual(VM* vm UNUSED, Value* args) {
562     if (!validateNum(vm, args[1])) {
563         RET_TRUE;
564     }
```




```
565      RET_BOOL(VALUE_TO_NUM(args[0]) != VALUE_TO_NUM(args[1]));
566 }
...略
```

函数 primNumPi 用来返回圆周率 3.14159265358979323846。

接下来要实现的函数大部分都是中缀运算符的原生方法，它们的实现具有共性：先要校验数字的合法性，通过后再用 args[0]和 args[1]做中缀运算符表示的运算。比如对于表达式 1+2 来说，args[0]是 1，args[1]是 2，中缀运算符是+，那么此表达式的计算公式即 args[0] operator args[1]。因此定义宏 PRIM_NUM_INFIX 来实现这些函数。下面第 435～442 行生成了大部分中缀运算符对应的原生方法。

第 445 行的宏 PRIM_NUM_BIT 用来实现数字之间“位操作”的原生方法，逻辑同 PRIM_NUM_INFIX 类似，很简单不说了，然后第 455～458 行调用此宏生成了 4 个位运算符原生方法。

第 462 行的宏 PRIM_NUM_MATH_FN 用于实现脚本中数学库函数对应的原生方法，第 467～477 行利用此宏生成了大部分的数学方法。

接下来定义的大部分原生方法都比较简单了，大家自己看吧。

stepByStep/c8/e/vm/core.c (3/3)

```
788  //绑定 num 类方法
789    vm->numClass = VALUE_TO_CLASS(getCoreClassValue(coreModule, "Num"));
790  //类方法
791  PRIM_METHOD_BIND(vm->numClass->objHeader.class, "fromString(_)",
     primNumFromString);
792        PRIM_METHOD_BIND(vm->numClass->objHeader.class, "pi", primNumPi);
793  //实例方法
794  PRIM_METHOD_BIND(vm->numClass, "+(_)", primNumPlus);
795  PRIM_METHOD_BIND(vm->numClass, "-(_)", primNumMinus);
796  PRIM_METHOD_BIND(vm->numClass, "*(_)", primNumMul);
797  PRIM_METHOD_BIND(vm->numClass, "/(_)", primNumDiv);
798  PRIM_METHOD_BIND(vm->numClass, ">(_)", primNumGt);
799  PRIM_METHOD_BIND(vm->numClass, ">=(_)", primNumGe);
800  PRIM_METHOD_BIND(vm->numClass, "<(_)", primNumLt);
801  PRIM_METHOD_BIND(vm->numClass, "<=(_)", primNumLe);
802
803  //位运算
804  PRIM_METHOD_BIND(vm->numClass, "&(_)", primNumBitAnd);
805  PRIM_METHOD_BIND(vm->numClass, "|(_)", primNumBitOr);
806  PRIM_METHOD_BIND(vm->numClass, ">>(_)", primNumBitShiftRight);
807  PRIM_METHOD_BIND(vm->numClass, "<<(_)", primNumBitShiftLeft);
808  //以上都是通过 rules 中 INFIX_OPERATOR 来解析的
809
810  //下面大多数方法是通过 rules 中.对应的 led(callEntry)来解析,
811  //少数符号依然是 INFIX_OPERATOR 解析
812  PRIM_METHOD_BIND(vm->numClass, "abs", primNumAbs);
813  PRIM_METHOD_BIND(vm->numClass, "acos", primNumAcos);
814  PRIM_METHOD_BIND(vm->numClass, "asin", primNumAsin);
815  PRIM_METHOD_BIND(vm->numClass, "atan", primNumAtan);
816  PRIM_METHOD_BIND(vm->numClass, "ceil", primNumCeil);
817  PRIM_METHOD_BIND(vm->numClass, "cos", primNumCos);
818  PRIM_METHOD_BIND(vm->numClass, "floor", primNumFloor);
819  PRIM_METHOD_BIND(vm->numClass, "-", primNumNegate);
```

```
820  PRIM_METHOD_BIND(vm->numClass, "sin", primNumSin);
821  PRIM_METHOD_BIND(vm->numClass, "sqrt", primNumSqrt);
822  PRIM_METHOD_BIND(vm->numClass, "tan", primNumTan);
823  PRIM_METHOD_BIND(vm->numClass, "%(_)", primNumMod);
824  PRIM_METHOD_BIND(vm->numClass, "~", primNumBitNot);
825  PRIM_METHOD_BIND(vm->numClass, "..(_)", primNumRange);
826  PRIM_METHOD_BIND(vm->numClass, "atan(_)", primNumAtan2);
827  PRIM_METHOD_BIND(vm->numClass, "fraction", primNumFraction);
828  PRIM_METHOD_BIND(vm->numClass, "isInfinity", primNumIsInfinity);
829  PRIM_METHOD_BIND(vm->numClass, "isInteger", primNumIsInteger);
830  PRIM_METHOD_BIND(vm->numClass, "isNan", primNumIsNan);
831  PRIM_METHOD_BIND(vm->numClass, "toString", primNumToString);
832  PRIM_METHOD_BIND(vm->numClass, "truncate", primNumTruncate);
833  PRIM_METHOD_BIND(vm->numClass, "==(_)", primNumEqual);
834  PRIM_METHOD_BIND(vm->numClass, "!=(_)", primNumNotEqual);
835 }
```

这部分代码还是之前的方法，把脚本方法与相应的原生方法绑定关联。

8.6 String 类及其方法（本节源码 stepByStep/c8/f）

字符串也是对象，也有一些内建的字符串方法，以下是示例代码。

```
[work@work spr]$ cat -n /tmp/test.sp
     1var str = "hello,"+"world"
     2System.print(str)
     3System.print(str.count)
     4System.print("hello".startsWith("he"))
     5System.print("hello".contains("ell"))
     6System.print("hello".contains("world"))
     7System.print("world"[0])
     8System.print("world"[-1..0])
     9System.print("world"[-1..-3])
[work@work spr]$ ./spr /tmp/test.sp
hello,world
11
true
true
false
w
dlrow
dlr
[work@work spr]$
```

以上字符串变量 str 不是必需的，只是演示。

本节继续耕耘文件 core.c，还是老方法，实现原生方法及一些基础，最后再通过宏 PRIM_METHOD_BIND 把脚本方法与原生方法绑定。

stepByStep/c8/f/vm/core.c(1/3)

```
...略
 134 //确认 value 是否为整数
```




```
135 static bool validateIntValue(VM* vm, double value) {
136     if (trunc(value) == value) {
137         return true;
138     }
139     SET_ERROR_FALSE(vm, "argument must be integer!");
140 }
141
142 //校验 arg 是否为整数
143 static bool validateInt(VM* vm, Value arg) {
144     //首先得是数字
145     if (!validateNum(vm, arg)) {
146         return false;
147     }
148
149     //再校验数值
150     return validateIntValue(vm, VALUE_TO_NUM(arg));
151 }
152
153 //校验参数 index 是否是落在[0, length)之间的整数
154 static uint32_t validateIndexValue(VM* vm, double index, uint32_t length) {
155     //索引必须是数字
156     if (!validateIntValue(vm, index)) {
157         return UINT32_MAX;
158     }
159
160     //支持负数索引,负数是从后往前索引
161     //转换其对应的正数索引，如果校验失败则返回 UINT32_MAX
162     if (index < 0) {
163         index += length;
164     }
165
166     //索引应该落在[0,length)
167     if (index >= 0 && index < length) {
168         return (uint32_t)index;
169     }
170
171     //执行到此说明超出范围
172     vm->curThread->errorObj =
173        OBJ_TO_VALUE(newObjString(vm, "index out of bound!", 19));
174     return UINT32_MAX;
175 }
176
177 //验证 index 有效性
178 static uint32_t validateIndex(VM* vm, Value index, uint32_t length) {
179     if (!validateNum(vm, index)) {
180         return UINT32_MAX;
181     }
182     return validateIndexValue(vm, VALUE_TO_NUM(index), length);
183 }
184
```

```
185 //从码点 value 创建字符串
186 static Value makeStringFromCodePoint(VM* vm, int value) {
187     uint32_t byteNum = getByteNumOfEncodeUtf8(value);
188     ASSERT(byteNum != 0, "utf8 encode bytes should be between 1 and 4!");
189
190     //+1 是为了结尾的\0
191     ObjString* objString = ALLOCATE_EXTRA(vm, ObjString, byteNum + 1);
192
193     if (objString == NULL) {
194         MEM_ERROR("allocate memory failed in runtime!");
195     }
196
197     initObjHeader(vm, &objString->objHeader, OT_STRING, vm->stringClass);
198     objString->value.length = byteNum;
199     objString->value.start[byteNum] = '\0';
200     encodeUtf8((uint8_t*)objString->value.start, value);
201     hashObjString(objString);
202     return OBJ_TO_VALUE(objString);
203 }
204
205 //用索引 index 处的字节创建字符串对象
206 static Value stringCodePointAt(VM* vm, ObjString* objString, uint32_t index) {
207     ASSERT(index < objString->value.length, "index out of bound!");
208     int codePoint = decodeUtf8((uint8_t*)objString->value.start + index,
209             objString->value.length - index);
210
211     //若不是有效的 UTF-8 序列,就将其处理为单个裸字符
212     if (codePoint == -1) {
213         return OBJ_TO_VALUE(newObjString(vm, &objString->value.start[index], 1));
214     }
215
216     return makeStringFromCodePoint(vm, codePoint);
217 }
218
```

第 135 行定义的 validateIntValue 函数并不是原生方法（原生方法的方法名是以 prim 开头），是被后面众多原生方法需要的函数，它用于确认数字 value 是否为整数，注意，前提是 value 已经是数字。判断原理很简单，调用 trunc 函数将 value“斩断”取整数部分，若结果与 value 相等则说明 value 为整数。

函数 validateInt 用于校验 arg 是否为整数，与 validateIntValue 不同的是，arg 的类型未必是数字，因此先调用 validateNum 验证其是否为数字，然后调用 validateIntValue 验证其是否为整数。

函数 validateIndexValue 用于校验参数 index 是否是属于[0, length]之间的整数，注意这是正数区分，包括 0 和 length。编译型语言（正如 C 语言）不如脚本语言灵活，脚本语言为用户提供了更多的自由度，我们用“笨拙的”编译型语言实现“灵活的”脚本语言，肯定要把困难留给自己了。脚本语言对序列结构支持负数索引，编译型语言中的序列（如数组）正常情况下不支持负数索引（指针除外），因此我们要把用户所键入的负数索引转换为正数索引后才能用编译型语言来实现相应功能。索引肯定得是整数，本函数开头先调用 validateIntValue 校验 index 是否为整数，如

果不是则返回 UINT32_MAX（一会儿说下 UINT32_MAX）。然后通过将 index 加上 length，使 index 变成对应的正数索引，再后面就是验证，不说了。

UINT32_MAX 是 32 位无符号数的最大值，即 4294967295，用十六进制表示的话，就是 0xFFFFFFFF。大家应该有印象，对于有符号数，最高位为 1 则表示负数，那么对于纯粹的二进制序列 0xFFFFFFFF 来说，如果要把它看成无符号数（自然数），那么它就是 4294967295，如果要把它看成有符号数，那么它就是-1。可能直接说这么“原始”的概念大家都有点不明白，那我们简单点儿用 0xFF 来举例，0xFF 的二进制是 11111111，对于无符号来说，其值为 255，这个很好理解。若把 0xFF 当成有符号数，它的值是多少呢？也许有读者会想当然地误以为是-127，因为我们被很多教材误导过，最高位为符号位，最左边的 1 表示负号，右边的 7 个 1 表示 127，因此误认为就是-127，这是绝对错误的。是这样的，最高位除了起到符号位的作用之外，它还是数值的一部分，也就是说符号位不只是影响正负号。那么若把 0xFF 当成负数，其数值到底是多少？很简单，我们知道“正数+负数=0”。因此只要用 0 减去负数就得到正数，即负数的绝对值，只要加个负号就行了。0 减去 0xFF 的结果是 1，因此 0xFF 是-1。那么对于有符号数 0x80～0xFF（即二进制 10000000～11111111），其范围即是-128～-1。正负数是 0 的互补，互为补码，因此这个算法就相当于求补码，补码=反码+1。而反码就是各位取反。既然计算机把 0xFF 既当成 255 又当成-1，那么我们（人）到底把它当成什么呢？这取决于我们怎么看待它。如果把它当成有符号数，那么与其进行运算的也得是有符号数，同理，如果把它当成无符号数，那么与其进行运算的也得是无符号数才行，总之无论当成哪种形式，其结果的二进制序列都是一致的。比如 0xF1（二进制 11110001）既是无符号数的 241，又是有符号数的-15。我们对 0xF1 执行自加 1 的操作，这是用 inc 指令实现的，inc 指令执行后结果是 11110010，换成十进制既是 242，又是-14。大多数指令不需要考虑操作数是否有符号，是通用的，不过有些个别指令还是要单独对待两类操作数，比如 div 指令用于无符号数的除法，而 idiv 指令用于有符号数的除法。

再回来看 UINT32_MAX，它包含在头文件 stdint.h 中， UINT32_MAX 既是 0xFFFFFFFF（十进制 4294967295）又是-1，用代码验证一下。

```
[work@work tmp]$ cat -n 0.c
     1 #include <stdio.h>
     2 #include <stdint.h>
     3
     4 void main() {
     5     if (-1 == UINT32_MAX) {
     6         printf("%d\n", UINT32_MAX);
     7     }
     8
     9     if (4294967295 == UINT32_MAX) {
    10         printf("%u\n", UINT32_MAX);
    11     }
    12     }
[work@work tmp]$ gcc 0.c
[work@work tmp]$ ./a.out
-1
4294967295
[work@work tmp]$
```

不指定输出的文件名，gcc 默认用 a.out。验证之后还是那句话，当成无符号数还是有符号数，取决于看它的视角，视角一致就可以了。



解释了这么多其实就是想说可以把 UINT32_MAX 当成-1 来用，-1 通常是用于函数返回的失败通知，那么我们这里为什么不直接用-1 呢？原因是：首先对于索引来说，用户不可能会到第 4294967295 个索引元素，其次为避免把-1 当成负数索引，所以采用 UINT32_MAX 表示-1。

函数 validateIndex 用于验证 index 有效性，很简单不说了。

函数 makeStringFromCodePoint 用于从码点 value 中创建字符串。我们在开头先调用 getByteNumOfEncodeUtf8 获取 value，按照 UTF-8 编码所需要的字节数 byteNum，接下来创建字符串 objString。随后是一些初始化工作，然后调用函数 encodeUtf8 把 value 按照 UTF-8 编码后存储到 objString->value.start，随后调用 hashObjString 生成哈希值，最后返回该对象 objString。

函数 stringCodePointAt 的功能是用字符串 objString 中索引 index 处的字节创建字符串对象，index 必须指向 UTF-8 起始的高字节，否则若指向了 UTF-8 序列的中间部分就认为编码被“斩断”，因此返回-1。调用 decodeUtf8 获取 index 处字符的码点，如果返回值 codePoint 为-1，就说明位于 index 处的字节并不是指向 UTF-8 编码的高字节，也就是说该编码序列无效，因此将其按照单个字符来处理并返回。如果是有效的 UTF-8 编码序列，则调用 makeStringFromCodePoint 为其生成字符串并返回。

stepByStep/c8/f/vm/core.c(2/3)

```
219 //计算 objRange 中元素的起始索引及索引方向
220 static uint32_t calculateRange(VM* vm,
221           ObjRange* objRange, uint32_t* countPtr, int* directionPtr) {
222
223       uint32_t from = validateIndexValue(vm, objRange->from, *countPtr);
224       if (from == UINT32_MAX) {
225           return UINT32_MAX;
226       }
227
228       uint32_t to = validateIndexValue(vm, objRange->to, *countPtr);
229       if (to == UINT32_MAX) {
230           return UINT32_MAX;
231       }
232
233       //如果 from 和 to 为负值,经过 validateIndexValue 已经变成了相应的正索引
234       *directionPtr = from < to ? 1 : -1;
235       *countPtr = abs((int)(from - to)) + 1;
236       return from;
237 }
238
239 //以 UTF-8 编码从 sourceStr 中起始为 startIndex,方向为 direction 的 count 个字符创建字符串
240 static ObjString* newObjStringFromSub(VM* vm, ObjString* sourceStr,
241           int startIndex, uint32_t count, int direction) {
242
243       uint8_t* source = (uint8_t*)sourceStr->value.start;
244       uint32_t totalLength = 0, idx = 0;
245
246       //计算 count 个 UTF-8 编码的字符总共需要的字节数,后面好申请空间
247       while (idx < count) {
248           totalLength += getByteNumOfDecodeUtf8(source[startIndex + idx * direction]);
249           idx++;
250       }
```




```
251
252     //+1 是为了结尾的\0
253     ObjString* result = ALLOCATE_EXTRA(vm, ObjString, totalLength + 1);
254
255     if (result == NULL) {
256         MEM_ERROR("allocate memory failed in runtime!");
257     }
258     initObjHeader(vm, &result->objHeader, OT_STRING, vm->stringClass);
259     result->value.start[totalLength] = '\0';
260     result->value.length = totalLength;
261
262     uint8_t* dest = (uint8_t*)result->value.start;
263     idx = 0;
264     while (idx < count) {
265         int index = startIndex + idx * direction;
266         //解码,获取字符数据
267         int codePoint = decodeUtf8(source + index, sourceStr->value.length -
          index);
268         if (codePoint != -1) {
269             //再将数据按照 UTF-8 编码,写入 result
270             dest += encodeUtf8(dest, codePoint);
271         }
272         idx++;
273     }
274
275     hashObjString(result);
276     return result;
277 }
278
279 //使用 Boyer-Moore-Horspool 字符串匹配算法在 haystack 中查找 needle,大海捞针
280 static int findString(ObjString* haystack, ObjString* needle) {
281     //如果待查找的 pattern 为空则为找到
282     if (needle->value.length == 0) {
283         return 0;       //返回起始下标 0
284     }
285
286     //若待搜索的字符串比原串还长，肯定搜不到
287     if (needle->value.length > haystack->value.length) {
288         return -1;
289     }
290
291     //构建“bad-character shift”表以确定窗口滑动的距离
292     //数组 shift 的值便是滑动距离
293     uint32_t shift[UINT8_MAX];
294     //needle 中最后一个字符的下标
295     uint32_t needleEnd = needle->value.length - 1;
296
297     //一、 先假定 bad character 不属于 needle(即 pattern),
298     //对于这种情况,滑动窗口跨过整个 needle
299     uint32_t idx = 0;
```




```
    while (idx < UINT8_MAX) {
        // 默认为滑过整个 needle 的长度
        shift[idx] = needle->value.length;
        idx++;
    }

    //二、假定 haystack 中与 needle 不匹配的字符在 needle 中之前已匹配过的位置出现过
    //就滑动窗口以使该字符与在 needle 中匹配该字符的最末位置对齐
    //这里预先确定需要滑动的距离
    idx = 0;
    while (idx < needleEnd) {
        char c = needle->value.start[idx];
        //idx 从前往后遍历 needle，当 needle 中有重复的字符 c 时，
        //后面的字符 c 会覆盖前面的同名字符 c，这保证了数组 shift 中字符是 needle 中最末位置的字符，
        //从而保证了 shift[c] 的值是 needle 中最末端同名字符与 needle 末端的偏移量
        shift[(uint8_t)c] = needleEnd - idx;
        idx++;
    }

    //Boyer-Moore-Horspool 是从后往前比较，这是处理 bad-character 高效的地方，
    //因此获取 needle 中最后一个字符，用于同 haystack 的窗口中最后一个字符比较
    char lastChar = needle->value.start[needleEnd];

    //长度差便是滑动窗口的滑动范围
    uint32_t range = haystack->value.length - needle->value.length;

    //从 haystack 中扫描 needle，寻找第 1 个匹配的字符，如果遍历完了就停止
    idx = 0;
    while (idx <= range) {
        //拿 needle 中最后一个字符同 haystack 窗口的最后一个字符比较
        //(因为 Boyer-Moore-Horspool 是从后往前比较)，如果匹配，看整个 needle 是否匹配
        char c = haystack->value.start[idx + needleEnd];
        if (lastChar == c &&
                memcmp(haystack->value.start + idx, needle->value.start,
                needleEnd) == 0) {
            //找到了就返回匹配的位置
            return idx;
        }

        //否则就向前滑动继续下一轮比较
        idx += shift[(uint8_t)c];
    }

    //未找到就返回-1
    return -1;
}
```

函数 calculateRange 用于计算 objRange 中元素的起始索引及索引方向，形参 countPtr 是存储 objRange 所能索引的元素个数的变量的指针，directionPtr 表示索引方向。我们将在后面定义 range 类，现在提前用到了，原因是字符串对象在检索字符的时候支持两种索引，一种是数字索引，另



一种是范围索引，就是 range 对象，我们在样例代码中有例子，无论哪种索引都支持负数。索引方向指的是当用 range 对象从字符串中索引字符时，索引的变化是递增还是递减，range 就是一段用 from 作为起始，用 to 作为结束的范围，如果 from 小于 to，这就是递增的索引范围，此时方向为正方向，用 1 表示，否则为负方向用-1 表示。我们先调用函数 validateIndexValue 校验 range 对象的 from 和 to 的合法性，如果索引为负，函数 validateIndexValue 会把索引转换为相应的正索引值。随后判断 from 是否比 to 小，将结果 1 或-1 存储到指针 directionPtr 指向的变量，将索引到的元素个数存储到指针 countPtr 指向的变量，最后返回 from。

函数 newObjStringFromSub 功能是以 UTF-8 为编码从 sourceStr 中获取起始为 startIndex、方向为 direction 的 count 个字符创建字符串。函数开头使指针 source 指向源字符串对象 sourceStr 中字符串的起始，变量 totalLength 用于存储 UTF-8 编码总共的字节数，idx 用于索引字符，这两个都是为下面的 while 循环做准备的。接下来生成字符串对象 result 以存储结果字符串，然后是一些初始化工作。编码工作是从下面的 while 循环开始的，开始之前先使 dest 指向 result 中存储字符串的地址，随后在 while 循环中先调用 decodeUtf8 获得字符的码点，然后调用 encodeUtf8 将码点按照 UTF-8 编码并写入 dest 即 result。最后为 result 创建哈希码并返回。

函数 findString 采用 Boyer-Moore-Horspool 字符串匹配算法在 haystack 中查找 needle，形参名很形象，haystack 意思是干草垛、干草堆，needle 意思是针，从干草垛中找一根针——大海捞针。此函数的实现就不多说了，都是公开的算法，原理自行参考资料吧，结合注释帮助理解。

stepByStep/c8/f/vm/core.c(3/3)

```
...略
781 //objString.fromCodePoint(_):从码点建立字符串
782 static bool primStringFromCodePoint(VM* vm, Value* args) {
783
784     if (!validateInt(vm, args[1])) {
785         return false;
786     }
787
788     int codePoint = (int)VALUE_TO_NUM(args[1]);
789     if (codePoint < 0) {
790         SET_ERROR_FALSE(vm, "code point can't be negetive!");
791     }
792
793     if (codePoint > 0x10ffff) {
794         SET_ERROR_FALSE(vm, "code point must be between 0 and 0x10ffff!");
795     }
796
797     RET_VALUE(makeStringFromCodePoint(vm, codePoint));
798 }
799
800 //objString+objString: 字符串相加
801 static bool primStringPlus(VM* vm, Value* args) {
802     if (!validateString(vm, args[1])) {
803         return false;
804     }
805
806     ObjString* left = VALUE_TO_OBJSTR(args[0]);
807     ObjString* right = VALUE_TO_OBJSTR(args[1]);
```

```
808
809     uint32_t totalLength = strlen(left->value.start) + strlen(right->value.start);
810     //+1 是为了结尾的\0
811     ObjString* result = ALLOCATE_EXTRA(vm, ObjString, totalLength + 1);
812     if (result == NULL) {
813         MEM_ERROR("allocate memory failed in runtime!");
814     }
815     initObjHeader(vm, &result->objHeader, OT_STRING, vm->stringClass);
816     memcpy(result->value.start, left->value.start, strlen(left->value.start));
817     memcpy(result->value.start + strlen(left->value.start),
818         right->value.start, strlen(right->value.start));
819     result->value.start[totalLength] = '\0';
820     result->value.length = totalLength;
821     hashObjString(result);
822
823     RET_OBJ(result);
824 }
825
826 //objString[_]:用数字或 objRange 对象做字符串的 subscript
827 static bool primStringSubscript(VM* vm, Value* args) {
828     ObjString* objString = VALUE_TO_OBJSTR(args[0]);
829     //数字和 objRange 都可以做索引,分别判断
830     //若索引是数字，就直接索引 1 个字符,这是最简单的 subscript
831     if (VALUE_IS_NUM(args[1])) {
832                 uint32_t index = validateIndex(vm, args[1],
                    objString->value.length);
833         if (index == UINT32_MAX) {
834             return false;
835         }
836         RET_VALUE(stringCodePointAt(vm, objString, index));
837     }
838
839     //索引要么为数字，要么为 ObjRange,若不是数字就应该为 objRange
840     if (!VALUE_IS_OBJRANGE(args[1])) {
841         SET_ERROR_FALSE(vm, "subscript should be integer or range!");
842     }
843
844     //direction 是索引的方向,
845     //1 表示正方向,从前往后，-1 表示反方向,从后往前
846     //from 若比 to 大,即从后往前检索字符,direction 则为-1
847     int direction;
848
849     uint32_t count = objString->value.length;
850     //返回的 startIndex 是 objRange.from 在 objString.value.start 中的下标
851             uint32_t startIndex = calculateRange(vm, VALUE_TO_OBJRANGE
                (args[1]), &count, &direction);
852     if (startIndex == UINT32_MAX) {
853         return false;
854     }
855
```

```
856         RET_OBJ(newObjStringFromSub(vm, objString, startIndex, count, direction));
857 }
858
859 //objString.byteAt_():返回指定索引的字节
860 static bool primStringByteAt(VM* vm UNUSED, Value* args) {
861     ObjString* objString = VALUE_TO_OBJSTR(args[0]);
862     uint32_t index = validateIndex(vm, args[1], objString->value.length);
863     if (index == UINT32_MAX) {
864         return false;
865     }
866     //故转换为数字返回
867     RET_NUM((uint8_t)objString->value.start[index]);
868 }
869
870 //objString.byteCount_:返回字节数
871 static bool primStringByteCount(VM* vm UNUSED, Value* args) {
872     RET_NUM(VALUE_TO_OBJSTR(args[0])->value.length);
873 }
874
875 //objString.codePointAt_(_):返回指定的 CodePoint
876 static bool primStringCodePointAt(VM* vm UNUSED, Value* args) {
877     ObjString* objString = VALUE_TO_OBJSTR(args[0]);
878     uint32_t index = validateIndex(vm, args[1], objString->value.length);
879     if (index == UINT32_MAX) {
880         return false;
881     }
882
883     const uint8_t* bytes = (uint8_t*)objString->value.start;
884     if ((bytes[index] & 0xc0) == 0x80) {
885         //如果 index 指向的并不是 UTF-8 编码的最高字节
886         //而是后面的低字节,返回-1 提示用户
887         RET_NUM(-1);
888     }
889
890     //返回解码
891     RET_NUM(decodeUtf8((uint8_t*)objString->value.start + index,
892             objString->value.length - index));
893 }
894
895 //objString.contains(_):判断字符串 args[0]中是否包含子字符串 args[1]
896 static bool primStringContains(VM* vm UNUSED, Value* args) {
897     if (!validateString(vm, args[1])) {
898         return false;
899     }
900
901     ObjString* objString = VALUE_TO_OBJSTR(args[0]);
902     ObjString* pattern = VALUE_TO_OBJSTR(args[1]);
903     RET_BOOL(findString(objString, pattern) != -1);
904 }
905
```

```
906 //objString.endsWith(_): 返回字符串是否以 args[1]为结束
907 static bool primStringEndsWith(VM* vm UNUSED, Value* args) {
908     if (!validateString(vm, args[1])) {
909         return false;
910     }
911
912     ObjString* objString = VALUE_TO_OBJSTR(args[0]);
913     ObjString* pattern = VALUE_TO_OBJSTR(args[1]);
914
915     //若 pattern 比源串还长,源串必然不包括 pattern
916     if (pattern->value.length > objString->value.length) {
917         RET_FALSE;
918     }
919
920     char* cmpIdx = objString->value.start +
921        objString->value.length - pattern->value.length;
922                 RET_BOOL(memcmp(cmpIdx, pattern->value.start,
                    pattern->value.length) == 0);
923 }
924
925 //objString.indexOf(_):检索字符串 args[0]中子串 args[1]的起始下标
926 static bool primStringIndexOf(VM* vm UNUSED, Value* args) {
927     if (!validateString(vm, args[1])) {
928         return false;
929     }
930
931     ObjString* objString = VALUE_TO_OBJSTR(args[0]);
932     ObjString* pattern = VALUE_TO_OBJSTR(args[1]);
933
934     //若 pattern 比源串还长,源串必然不包括 pattern
935     if (pattern->value.length > objString->value.length) {
936         RET_FALSE;
937     }
938
939     int index = findString(objString, pattern);
940     RET_NUM(index);
941 }
942
943 //objString.iterate(_):返回下一个 UTF-8 字符(不是字节)的迭代器
944 static bool primStringIterate(VM* vm UNUSED, Value* args) {
945     ObjString* objString = VALUE_TO_OBJSTR(args[0]);
946
947     //如果是第一次迭代，迭代索引肯定为空
948     if (VALUE_IS_NULL(args[1])) {
949         if (objString->value.length == 0) {
950             RET_FALSE;
951         }
952         RET_NUM(0);
953     }
954
```

```
955       //迭代器必须是正整数
956       if (!validateInt(vm, args[1])){
957            return false;
958       }
959
960       double iter = VALUE_TO_NUM(args[1]);
961       if (iter < 0) {
962           RET_FALSE;
963       }
964
965       uint32_t index = (uint32_t)iter;
966       do {
967          index++;
968
969          //到了结尾就返回 false,表示迭代完毕
970          if (index >= objString->value.length) RET_FALSE;
971
972          //读取连续的数据字节,直到下一个 UTF-8 的高字节
973       } while ((objString->value.start[index] & 0xc0) == 0x80);
974
975       RET_NUM(index);
976 }
977
978 //objString.iterateByte_(_): 迭代索引,内部使用
979 static bool primStringIterateByte(VM* vm UNUSED, Value* args) {
980       ObjString* objString = VALUE_TO_OBJSTR(args[0]);
981
982       //如果是第一次迭代，迭代索引肯定为空，直接返回索引 0
983       if (VALUE_IS_NULL(args[1])) {
984            if (objString->value.length == 0) {
985                 RET_FALSE;
986            }
987            RET_NUM(0);
988       }
989
990       //迭代器必须是正整数
991       if (!validateInt(vm, args[1])) {
992            return false;
993       }
994
995       double iter = VALUE_TO_NUM(args[1]);
996
997       if (iter < 0) {
998           RET_FALSE;
999       }
1000
1001      uint32_t index = (uint32_t)iter;
1002      index++; //移进到下一个字节的索引
1003      if (index >= objString->value.length) {
1004           RET_FALSE;
```

```
1005        }
1006
1007        RET_NUM(index);
1008    }
1009
1010    //objString.iteratorValue(_):返回迭代器对应的 value
1011    static bool primStringIteratorValue(VM* vm, Value* args) {
1012        ObjString* objString = VALUE_TO_OBJSTR(args[0]);
1013        uint32_t index = validateIndex(vm, args[1], objString->value.length);
1014        if (index == UINT32_MAX) {
1015            return false;
1016        }
1017        RET_VALUE(stringCodePointAt(vm, objString, index));
1018    }
1019
1020    //objString.startsWith(_): 返回 args[0]是否以 args[1]为起始
1021    static bool primStringStartsWith(VM* vm UNUSED, Value* args) {
1022        if (!validateString(vm, args[1])) {
1023            return false;
1024        }
1025
1026        ObjString* objString = VALUE_TO_OBJSTR(args[0]);
1027        ObjString* pattern = VALUE_TO_OBJSTR(args[1]);
1028
1029        //若 pattern 比源串还长,源串必然不包括 pattern,
1030        //因此不可能以 pattern 为起始
1031        if (pattern->value.length > objString->value.length) {
1032            RET_FALSE;
1033        }
1034
1035        RET_BOOL(memcmp(objString->value.start,
1036                pattern->value.start, pattern->value.length) == 0);
1037    }
1038
1039    //objString.toString:获得自己的字符串
1040    static bool primStringToString(VM* vm UNUSED, Value* args) {
1041        RET_VALUE(args[0]);
1042    }
…略
1312        //字符串类
1313        vm->stringClass = VALUE_TO_CLASS(getCoreClassValue(coreModule, "String"));
1314                    PRIM_METHOD_BIND(vm->stringClass->objHeader.class,
                        "fromCodePoint(_)", primStringFromCodePoint);
1315        PRIM_METHOD_BIND(vm->stringClass, "+(_)", primStringPlus);
1316        PRIM_METHOD_BIND(vm->stringClass, "[_]", primStringSubscript);
1317        PRIM_METHOD_BIND(vm->stringClass, "byteAt_(_)", primStringByteAt);
1318        PRIM_METHOD_BIND(vm->stringClass, "byteCount_", primStringByteCount);
1319        PRIM_METHOD_BIND(vm->stringClass, "codePointAt_(_)", primStringCodePointAt);
1320        PRIM_METHOD_BIND(vm->stringClass, "contains(_)", primStringContains);
1321        PRIM_METHOD_BIND(vm->stringClass, "endsWith(_)", primStringEndsWith);
```

```
1322        PRIM_METHOD_BIND(vm->stringClass, "indexOf(_)", primStringIndexOf);
1323        PRIM_METHOD_BIND(vm->stringClass, "iterate(_)", primStringIterate);
1324        PRIM_METHOD_BIND(vm->stringClass, "iterateByte_(_)", primStringIterateByte);
1325        PRIM_METHOD_BIND(vm->stringClass, "iteratorValue(_)", primStringIteratorValue);
1326        PRIM_METHOD_BIND(vm->stringClass, "startsWith(_)", primStringStartsWith);
1327        PRIM_METHOD_BIND(vm->stringClass, "toString", primStringToString);
1328        PRIM_METHOD_BIND(vm->stringClass, "count", primStringByteCount);
1329 }
```

这第 3 部分代码是原生方法。

函数 primStringFromCodePoint 对应于脚本中的方法 objString.fromCodePoint(args[1])，用于从码点 args[1]建立字符串。函数开头在 3 个方面校验码点 args[1]，首先它应该为正整数，其次它应该大于等于 0，最后不能大于 0x10ffff。满足这 3 个条件后，调用 makeStringFromCodePoint 从码点 args[1]创建字符串。

函数 primStringPlus 用于两个字符串相加，如"foo"+"bar"便会由本函数处理。函数开头先分别获得+的左右操作数（如"foo"为左操作数，"bar"为右操作数），然后计算相加后的字符串的总长度，接下来用总长度加 1 作为 ALLOCATE_EXTRA 的参数为结果字符串 result 申请空间。随后就是调用 memcpy 往 result->value.start 中拷贝左右操作数的字符串，最后返回结果 result。

函数 primStringSubscript 用于索引字符串，数字和 objRange 对象都可作为索引，数字只能索引 1 个字符，而 objRange 对象是个范围，因此可以索引一段连续的字符。比如"abcde"[3]是用数字做索引，值为 d，"abcde"[2:4]是用 objRange 做索引，值为 cde。args[0]是被索引的字符串，args[1]是索引。先调用宏 VALUE_TO_OBJSTR 获得字符串对象 objString，若是数字作为索引，那么还要通过函数 validateIndex 校验此数字是否在[0～objString->value.length]之间（开区间表示不包括 objString->value.length），若符合要求就调用 stringCodePointAt 为该索引处的字符生成字符串对象并返回。如果索引不是数字，到了第 840 行，索引就只能是 objRange 对象，如果不是则报错。direction 用于表示索引的方向，注释也很清楚，不再赘述。下面是 count 变量的值，count 目前是 objString 中字符的总数，但 calculateRange 返回之后，count 会被调整为 range 中 from 和 to 之间的个数。提醒一下，传给 calculateRange 的 count 是指针，calculateRange 会根据 objRange 中 from 和 to 的取值范围对 count 有所取舍。返回值 startIndex 是被索引的字符串的起始下标，最后调用 newObjStringFromSub 从字符串 objString 中获取起始为 startIndex、方向为 direction 的 count 个字符创建字符串并返回。

函数 primStringByteAt 对应于方法 objString.byteAt_()，以下划线结尾的方法仅供内部使用，当然如果脚本中直接调用，那么编译上也没做限制，此函数功能用于返回指定索引 args[1]的字节，实现比较简单，不说了。

函数 primStringByteCount 是 objString.byteCount_的原生方法，也是供内部使用，用于返回字节数，注意是字节数，并不是字符数，因为这涉及多字节字符编码。函数就是直接返回 objString->value.length。

函数 primStringCodePointAt 对应于脚本方法 objString.codePointAt_(_)，用于返回 args[1]指定的 CodePoint。先通过 VALUE_TO_OBJSTR 将 args[0]转换为 objString，然后调用 validateIndex 校验并转换（如果索引为负）索引 args[1]。接下来判断索引指向的字节是否是 UTF-8 的高字节，如果不是就说明 UTF-8 编码被斩断，或者编码序列是错误的，毕竟只有指向 UTF-8 高字节时我们才知道后面有几个用于存储码点的字节，因此若不是指向高字节我们就返回-1。验证通过后我们就调用 decodeUtf8 解码并返回。

函数 primStringContains 是脚本方法 objString.contains(_)的原生方法，用于判断字符串 args[0]中是否包含子字符串 args[1]。核心思路就是调用 findString 去查找。

函数 primStringEndsWith 是脚本方法 objString.endsWith(_)的原生方法，用于返回字符串是否以 args[1]为结束。校验 args[1]之后，获取源串到 objString，获取目标串 pattern，接下来对比两个串的长度，若 pattern 比源串还长，则源串必然不包括 pattern，因此返回 false。否则调用 memcpy 去比较并返回结果。

函数 primStringIndexOf 对应脚本方法 objString.indexOf(_)，用于检索字符串 args[0]中子串 args[1]的起始下标，核心是调用 findString 从 arsg[0]中查找 args[1]，很简单，不说了。

函数 primStringIterate 对应脚本方法 objString.iterate(_)，用于返回下一个 UTF-8 字符（不是字节）的迭代器，即用来迭代字符。先把 args[0]转换为字符串对象 objString，接下来判断迭代器参数 args[1]是否为空，因为第一次迭代时 args[1]为 null，因此应该返回 0，即第 0 个元素。后面继续验证迭代器应该为正整数。下面的 do-while 循环中的条件目的是向前移进到下一个 UTF-8 字符的起始。UTF-8 字符是 1～4 个字节，数据部分的字节以 10 开头，故与 0xc0 相与后结果是 0x80。连续读取多个数据字节后便是下一个 UTF-8 字符的最高字节，高字节的标志是以两个以上的 1 开头并后接一个 0。1 个字节的 UTF-8 字符是以 0 开头的 ASCII 码，即小于等于 0x7f 的 UTF-8 码点就是 ASCII。由于 ASCII 字符只是 1 字节，其后没有 10 开头的数据字节，故采用 do-while，先执行循环体再判断条件。迭代器就是返回参数所指定索引的字符后面的字符的索引，这里所说的是下一个字符的索引，并不是字节，因此返回的未必是 args[1]+1，除非是单字节的 ASCII 字符。循环执行 index++后就跳过了 args[1]所指向的 UTF-8 编码的字符，如果 index 大于等于字符串 objString 的长度，则说明迭代完毕，因此返回 false。最后循环体外返回的 index 便是指向下一个 UTF-8 字符的高字节的索引，即下一个迭代器。

函数 primStringIterateByte 对应的方法 objString.iterateByte_(_)仅供内部使用，功能是迭代字节，注意，不是字符，因此本函数和 primStringIterate 的区别就是没有后面的 do-while 循环跳过多个 UTF-8 字符，实现类似，不说了。

函数 primStringIteratorValue 对应于脚本方法 objString.iteratorValue(_)，用于返回迭代器对应的 value，很简单大家自己看吧。

函数 primStringStartsWith 对应于脚本方法 objString.startsWith(_)，用于返回 args[0]是否以 args[1]为起始，即 bool 值。函数开头校验 args[1]为有效的字符串之后，分别获得源串 objString 即 args[0]和模式串 pattern args[1]，判断模式串不应该比源串长，最后调用 memcmp 依次比较开头的字符并返回结果。

函数 primStringToString 对应脚本方法 objString.toString，即输出自己。

最后还是老方法，把以上原生方法关联到脚本方法。说明一下，最后绑定的 count 方法只是返回字符串的字节数，并不是字符数，因此它与 byteCount_方法指向的是同一个原生方法 primStringByteCount。

8.7 List 类及其方法（本节源码 stepByStep/c8/g）

本节实现的是 List 类。以下是 List 类的样例代码。

```
[work@work spr]$ cat -n /tmp/test.sp
     1 var l = ["a", "b"]
     2 System.print("count:" + l.count.toString + " list:" +  l.toString)
     3 l.add("c")
```

```
 4 System.print("count:" + l.count.toString + " list:" +  l.toString)
 5 l.insert(1, "d")
 6 System.print("count:" + l.count.toString + " list:" +  l.toString)
 7 l.clear()
 8 System.print("count:" + l.count.toString + " list:" +  l.toString)
 9 var ln = List.new()
10 ln.add("a")
11 ln.add("b")
12 System.print(ln[0])
13 ln[0] = "c"
14 System.print(ln[0])
[work@work spr]$ ./spr /tmp/test.sp
count:2 list:[a,b]
count:3 list:[a,b,c]
count:4 list:[a,d,b,c]
count:0 list:[]
a
c
[work@work spr]$
```

List 类与前面 String 类及其他类的方法雷同，整体上也较简单，因此我只大体上介绍。

stepByStep/c8/g/vm.core.c

```
…略
1045 //objList.new():创建一个新的 list
1046 static bool primListNew(VM* vm, Value* args UNUSED) {
1047     RET_OBJ(newObjList(vm, 0));
1048 }
1049
1050 //objList[_]:索引 list 元素
1051 static bool primListSubscript(VM* vm, Value* args) {
1052     ObjList* objList = VALUE_TO_OBJLIST(args[0]);
1053
1054     //数字和 objRange 都可以做索引,分别判断
1055     //若索引是数字,就直接索引 1 个字符,这是最简单的 subscript
1056     if (VALUE_IS_NUM(args[1])) {
1057             uint32_t index = validateIndex(vm, args[1], objList->elements.count);
1058         if (index == UINT32_MAX) {
1059             return false;
1060         }
1061         RET_VALUE(objList->elements.datas[index]);
1062     }
1063
1064     //索引要么为数字要么为 ObjRange,若不是数字就应该为 objRange
1065     if (!VALUE_IS_OBJRANGE(args[1])) {
1066         SET_ERROR_FALSE(vm, "subscript should be integer or range!");
1067     }
1068
1069     int direction;
1070
1071     uint32_t count = objList->elements.count;
```

```
1072
1073      //返回的 startIndex 是 objRange.from 在 objList.elements.data 中的下标
1074      uint32_t startIndex = calculateRange(vm, VALUE_TO_OBJRANGE(args[1]), &count,
          &direction);
1075
1076      //新建一个 list 存储该 range 在原来 list 中索引的元素
1077      ObjList* result = newObjList(vm, count);
1078      uint32_t idx = 0;
1079      while (idx < count) {
1080         //direction 为-1 表示从后往前倒序赋值
1081         //如 var l = [a,b,c,d,e,f,g]; l[5..3]表示[f,e,d]
1082         result->elements.datas[idx] = objList->elements.datas[startIndex + idx
           * direction];
1083         idx++;
1084      }
1085      RET_OBJ(result);
1086 }
1087
1088 //objList[_]=(_):只支持数字作为 subscript
1089 static bool primListSubscriptSetter(VM* vm UNUSED, Value* args) {
1090      //获取对象
1091      ObjList* objList = VALUE_TO_OBJLIST(args[0]);
1092
1093      //获取 subscript
1094      uint32_t index = validateIndex(vm, args[1], objList->elements.count);
1095      if (index == UINT32_MAX) {
1096           return false;
1097      }
1098
1099      //直接赋值
1100      objList->elements.datas[index] = args[2];
1101
1102      RET_VALUE(args[2]); //把参数 2 作为返回值
1103 }
1104
1105 //objList.add(_):直接追加到 list 中
1106 static bool primListAdd(VM* vm, Value* args) {
1107      ObjList* objList = VALUE_TO_OBJLIST(args[0]);
1108      ValueBufferAdd(vm, &objList->elements, args[1]);
1109      RET_VALUE(args[1]); //把参数 1 作为返回值
1110 }
1111
1112 //objList.addCore_(_):编译内部使用的,用于编译列表直接量
1113 static bool primListAddCore(VM* vm, Value* args) {
1114      ObjList* objList = VALUE_TO_OBJLIST(args[0]);
1115      ValueBufferAdd(vm, &objList->elements, args[1]);
1116      RET_VALUE(args[0]); //返回列表自身
1117 }
1118
1119 //objList.clear():清空 list
```

```
1120 static bool primListClear(VM* vm, Value* args) {
1121     ObjList* objList = VALUE_TO_OBJLIST(args[0]);
1122     ValueBufferClear(vm, &objList->elements);
1123     RET_NULL;
1124 }
1125
1126 //objList.count:返回 list 中元素个数
1127 static bool primListCount(VM* vm UNUSED, Value* args) {
1128     RET_NUM(VALUE_TO_OBJLIST(args[0])->elements.count);
1129 }
1130
1131 //objList.insert(_,_):插入元素
1132 static bool primListInsert(VM* vm, Value* args) {
1133     ObjList* objList = VALUE_TO_OBJLIST(args[0]);
1134     //+1 确保可以在最后插入
1135     uint32_t index = validateIndex(vm, args[1], objList->elements.count + 1);
1136     if (index == UINT32_MAX) {
1137         return false;
1138     }
1139     insertElement(vm, objList, index, args[2]);
1140     RET_VALUE(args[2]);  //参数 2 作为返回值
1141 }
1142
1143 //objList.iterate(_):迭代 list
1144 static bool primListIterate(VM* vm, Value* args) {
1145     ObjList* objList = VALUE_TO_OBJLIST(args[0]);
1146
1147     //如果是第一次迭代，迭代索引肯定为空，直接返回索引 0
1148     if (VALUE_IS_NULL(args[1])) {
1149         if (objList->elements.count == 0) {
1150             RET_FALSE;
1151         }
1152         RET_NUM(0);
1153     }
1154
1155     //确保迭代器是整数
1156     if (!validateInt(vm, args[1])) {
1157         return false;
1158     }
1159
1160     double iter = VALUE_TO_NUM(args[1]);
1161     //如果迭代完了就终止
1162     if (iter < 0 || iter >= objList->elements.count - 1) {
1163         RET_FALSE;
1164     }
1165
1166     RET_NUM(iter + 1);   //返回下一个
1167 }
1168
1169 //objList.iteratorValue(_):返回迭代值
```

```
1170 static bool primListIteratorValue(VM* vm, Value* args) {
1171     //获取实例对象
1172     ObjList* objList = VALUE_TO_OBJLIST(args[0]);
1173
1174     uint32_t index = validateIndex(vm, args[1], objList->elements.count);
1175     if (index == UINT32_MAX) {
1176         return false;
1177     }
1178
1179     RET_VALUE(objList->elements.datas[index]);
1180 }
1181
1182 //objList.removeAt(_):删除指定位置的元素
1183 static bool primListRemoveAt(VM* vm, Value* args) {
1184     //获取实例对象
1185     ObjList* objList = VALUE_TO_OBJLIST(args[0]);
1186
1187     uint32_t index = validateIndex(vm, args[1], objList->elements.count);
1188     if (index == UINT32_MAX) {
1189         return false;
1190     }
1191
1192     RET_VALUE(removeElement(vm, objList, index));
1193 }
1194
…略
1481     //List 类
1482     vm->listClass = VALUE_TO_CLASS(getCoreClassValue(coreModule, "List"));
1483     PRIM_METHOD_BIND(vm->listClass->objHeader.class, "new()", primListNew);
1484     PRIM_METHOD_BIND(vm->listClass, "[_]", primListSubscript);
1485     PRIM_METHOD_BIND(vm->listClass, "[_]=(_)", primListSubscriptSetter);
1486     PRIM_METHOD_BIND(vm->listClass, "add(_)", primListAdd);
1487     PRIM_METHOD_BIND(vm->listClass, "addCore_(_)", primListAddCore);
1488     PRIM_METHOD_BIND(vm->listClass, "clear()", primListClear);
1489     PRIM_METHOD_BIND(vm->listClass, "count", primListCount);
1490     PRIM_METHOD_BIND(vm->listClass, "insert(_,_)", primListInsert);
1491     PRIM_METHOD_BIND(vm->listClass, "iterate(_)", primListIterate);
1492     PRIM_METHOD_BIND(vm->listClass, "iteratorValue(_)", primListIteratorValue);
1493     PRIM_METHOD_BIND(vm->listClass, "removeAt(_)", primListRemoveAt);
1494 }
```

函数 primListNew 对应于脚本方法 objList.new()，用于创建一个新的列表。直接调用 newObjList 生成 objList 对象返回。

函数 primListSubscript 对应于脚本方法 objList[_]，也就是索引 List 中的元素。先获得列表对象 objList。数字和 objRange 对象都可以作为索引，接下来判断索引即 args[1]，如果是数字，就调用 validateIndex 校验索引并转换为相应正索引（如果 args[1]为负），然后返回 objList 对象中相应元素。如果索引是 objRange 对象，需要调用 calculateRange 计算索引的范围，然后新建一个列表对象 result，将源列表对象 objList 中的元素拷贝到 result 中，最后返回 result。

函数 primListSubscriptSetter 对应于脚本方法 objList[_]=(_)，args[0]为待操作的列表对象，args[1]

为索引，args[2]为所赋的值，即 objList[args[1]]=args[2]，只支持数字作为索引。先获取列表对象 objList，再校验索引 args[1]并转换为 index（若 args[1]为负数），然后利用索引 index 修改指定的元素为 args[2]，最后返回 args[2]。

函数 primListAdd 对应于脚本方法 objList.add(_)，用于直接追加元素 args[1]到 objList 即 args[0]。函数就是用 ValueBufferAdd 往 objList 中添加 args[1]。

函数 primListAddCore 对应于脚本方法 objList.addCore_(_)，说过了，以下划线_结尾的是内部方法，同样用于往列表中添加元素，用于编译列表直接量，即不是用 List.new()而是直接用[“元素 1”，“元素 2”…]定义的列表。

函数 primListClear 对应于脚本方法 objList.clear()，用于清空列表，核心是调用 ValueBufferClear 实现列表元素数组 objList->elements 的清空。

函数 primListCount 对应于脚本方法 objList.count，用于返回 list 中元素个数，函数实现更简单，直接返回 elements.count。

函数 primListInsert 对应于脚本方法 objList.insert(_,_)，即在 objList 中位置 args[1]后插入元素 args[2]，args[0]就是 objList。这里是调用函数 insertElement 完成的插入（此函数是在 obj_list.c 中实现的），最后返回 args[2]。

函数 primListIterate 对应于脚本方法 objList.iterate(_)，用于迭代列表 objList，即 args[0]。和其他迭代函数类似，若是第一次迭代，传给脚本方法 iterate 的参数必为 null，因此就返回 0，即首个元素的索引。如果参数并不是 null，说明并不是第一次调用，因此要获得迭代器 args[1]，使其加 1 后返回作为下一个元素的迭代器。

函数 primListIteratorValue 对应于脚本方法 objList.iteratorValue(_)，用于返回迭代元素的值。先获取 objList 即 args[0]，然后校验并获取索引 index，直接返回 index 对应的值 objList->elements.datas [index]。

函数 primListRemoveAt 对应于脚本方法 objList.removeAt(_)，用于删除指定位置 args[1]的元素。先获取实例对象 objList，再获取待删除元素的索引 index，然后调用 removeElement（此函数是在 obj_list.c 中实现的）将元素删除。

接下来就是把原生方法与脚本方法绑定。

8.8 Map 类及其方法（本节源码 stepByStep/c8/h）

Map 就是我们常用的哈希数组，本节完成 Map 类方法的填充。样例代码如下。

```
[work@work spr]$ cat -n /tmp/test2.sp
     1 var m = {
     2     "a":1,
     3     "b":2,
     4     "c":3,
     5 }
     6
     7 for k (m.keys) {
     8     System.print(k + " -> " + m[k].toString)
     9 }
    10
    11 if (m.containsKey("c")) m["c"] = 4
    12 System.print(m.count)
```

仅供非商业用途或交流学习使用


```
    13 m.clear()
    14 System.print(m.count)
[work@work spr]$ ./spr /tmp/test2.sp
c -> 3
b -> 2
a -> 1
3
0
[work@work spr]$
```

注意，上例中第 7 行的 m.keys，其中的 keys 方法并没有对应的原生实现，而是在核心脚本 core.script.inc 中用脚本方法实现的，大家可以参考该文件。下面介绍原生方法。

stepByStep/c8/h/vm/core.c(1/2)

```
...略
 186 //校验 key 合法性
 187 static bool validateKey(VM* vm, Value arg) {
 188     if (VALUE_IS_TRUE(arg)       ||
 189         VALUE_IS_FALSE(arg)  ||
 190         VALUE_IS_NULL(arg)   ||
 191         VALUE_IS_NUM(arg)      ||
 192         VALUE_IS_OBJSTR(arg) ||
 193         VALUE_IS_OBJRANGE(arg)  ||
 194         VALUE_IS_CLASS(arg)) {
 195         return true;
 196     }
 197     SET_ERROR_FALSE(vm, "key must be value type!");
 198 }
...略
1209 //objMap.new():创建 map 对象
1210 static bool primMapNew(VM* vm, Value* args UNUSED) {
1211     RET_OBJ(newObjMap(vm));
1212 }
1213
1214 //objMap[_]:返回 map[key]对应的 value
1215 static bool primMapSubscript(VM* vm, Value* args) {
1216     //校验 key 的合法性
1217     if (!validateKey(vm, args[1])) {
1218         return false;  //出错了,切换线程
1219     }
1220
1221     //获得 map 对象实例
1222     ObjMap* objMap = VALUE_TO_OBJMAP(args[0]);
1223
1224     //从 map 中查找 key(args[1])对应的 value
1225     Value value = mapGet(objMap, args[1]);
1226
1227     //若没有相应的 key 则返回 NULL
1228     if (VALUE_IS_UNDEFINED(value)) {
1229         RET_NULL;
```

仅供非商业用途或交流学习使用


```
1230        }
1231
1232        RET_VALUE(value);
1233    }
1234
1235    //objMap[_]=(_):map[key]=value
1236    static bool primMapSubscriptSetter(VM* vm, Value* args) {
1237        //校验 key 的合法性
1238        if (!validateKey(vm, args[1])) {
1239            return false;  //出错了,切换线程
1240        }
1241
1242        //获得 map 对象实例
1243        ObjMap* objMap = VALUE_TO_OBJMAP(args[0]);
1244
1245        //在 map 中将 key 和 value 关联
1246        //即 map[key]=value
1247        mapSet(vm, objMap, args[1], args[2]);
1248
1249        RET_VALUE(args[2]); //返回 value
1250    }
1251
1252    //objMap.addCore_(_,_):编译器编译 map 字面量时内部使用的,
1253    //在 map 中添加(key-value)对儿并返回 map 自身
1254    static bool primMapAddCore(VM* vm, Value* args) {
1255        if (!validateKey(vm, args[1])) {
1256            return false;  //出错了,切换线程
1257        }
1258
1259        //获得 map 对象实例
1260        ObjMap* objMap = VALUE_TO_OBJMAP(args[0]);
1261
1262        //在 map 中将 key 和 value 关联
1263        //即 map[key]=value
1264        mapSet(vm, objMap, args[1], args[2]);
1265
1266        RET_VALUE(args[0]);  //返回 map 对象自身
1267    }
1268
1269    //objMap.clear():清除 map
1270    static bool primMapClear(VM* vm, Value* args) {
1271        clearMap(vm, VALUE_TO_OBJMAP(args[0]));
1272        RET_NULL;
1273    }
1274
1275    //objMap.containsKey(_):判断 map 即 args[0]是否包含 key 即 args[1]
1276    static bool primMapContainsKey(VM* vm, Value* args) {
1277        if (!validateKey(vm, args[1])) {
1278            return false;  //出错了,切换线程
1279        }
```

仅供非商业用途或交流学习使用


```
1280
1281      //直接去 get 该 key,判断是否 get 成功
1282      RET_BOOL(!VALUE_IS_UNDEFINED(mapGet(VALUE_TO_OBJMAP(args[0]), args[1])));
1283 }
1284
1285 //objMap.count:返回 map 中 entry 个数
1286 static bool primMapCount(VM* vm UNUSED, Value* args) {
1287      RET_NUM(VALUE_TO_OBJMAP(args[0])->count);
1288 }
1289
```

函数 validateKey 用于校验 arg 是否为合法的 Key 类型。这里大家看到了我们所支持的 key 包括 true、false、null、num、objString、objRange 和 Class。

接下来是原生函数，primMapNew 对应于脚本方法 objMap.new()，用于创建 map 对象。实现很简单，直接调用 newObjMap 生成 Map 对象并返回。

函数 primMapSubscript 对应于脚本方法 objMap[key]，用于返回 map[key]，即 key 对应的 value。调用 validateKey 检查 key 的合法性，若 key 合法，继续调用 VALUE_TO_OBJMAP 把 args[0]转换为 objMap 对象。然后调用 mapGet 从 objMap 中查找 key（即 args[1]）对应的 value，如果返回值 value 的类型并不是 UNDEFINED，说明该 key 存在，因此返回 value。

函数 primMapSubscriptSetter 对应于脚本方法 objMap[key]=(value)，即用 value 给 map 中的 key 赋值。此函数有 3 个参数；args[0]是 map 对象，args[1]是 key，args[2]是 value。前面步骤依然是校验 key 即 args[1]，然后获取 map 对象 objMap，不同的是这里要调用的是 mapSet，把 key 和 value 写入 objMap，最后返回 value 即 args[2]。

函数 primMapAddCore 对应于脚本方法 objMap.addCore_(key,value)，此函数用于处理 map 直接量定义，即不是用 objMap.new()方法创建，而是用大括号{key1:value1, key2:value2,...}定义的关联数组。先调用 validateKey 校验 key 是否合法，关联数组就是调用 mapSet 使 key 和 value 关联在一起，mapSet 是在 obj_map.c 中定义的。

函数 primMapClear 对应于脚本方法 objMap.clear()，用于清空 objMap。直接调用 clearMap 完成，clearMap 同样是定义在 obj_map.c 中。

函数 primMapContainsKey 对应于脚本方法 objMap.containsKey(_)，用于判断 objMap 即 args[0]是否包含 key 即 args[1]。在通过 validateKey 校验 key 即 args[1]合法后，直接用 mapGet 从 objMap 获取 key，如果返回值不是 UNDEFINED，说明该 key 存在。

函数 primMapCount 对应于 objMap.count，用于返回 map 中 entry 个数，函数实现很简单，直接返回 objMap->count 即可。

stepByStep/c8/h/vm/core.c(2/2)

```
1290 //objMap.remove(_):删除 map[key], map 是 args[0], key 是 args[1]
1291 static bool primMapRemove(VM* vm, Value* args) {
1292      if (!validateKey(vm, args[1])) {
1293           return false;  //出错了,切换线程
1294      }
1295
1296      RET_VALUE(removeKey(vm, VALUE_TO_OBJMAP(args[0]), args[1]));
1297 }
1298
1299 //objMap.iterate_(_):迭代 map 中的 entry,
```

仅供非商业用途或交流学习使用


```
1300 //返回 entry 的索引供 keyIteratorValue_和 valueIteratorValue_做迭代器
1301 static bool primMapIterate(VM* vm, Value* args) {
1302     //获得 map 对象实例
1303     ObjMap* objMap = VALUE_TO_OBJMAP(args[0]);
1304
1305     //map 中若空则返回 false, 不可迭代
1306     if (objMap->count == 0) {
1307         RET_FALSE;
1308     }
1309
1310     //若没有传入迭代器,迭代默认是从第 0 个 entry 开始
1311     uint32_t index = 0;
1312
1313     //若不是第一次迭代,传进了迭代器
1314     if (!VALUE_IS_NULL(args[1])) {
1315         //iter 必须为整数
1316         if (!validateInt(vm, args[1])) {
1317             //本线程出错了,返回 false 是为了切换到下一线
1318             return false;
1319         }
1320
1321         //迭代器不能小于 0
1322         if (VALUE_TO_NUM(args[1]) < 0) {
1323             RET_FALSE;
1324         }
1325
1326         index = (uint32_t)VALUE_TO_NUM(args[1]);
1327         //迭代器不能越界
1328         if (index >= objMap->capacity) {
1329             RET_FALSE;
1330         }
1331
1332         index++;  //更新迭代器
1333     }
1334
1335     //返回下一个正在使用(有效)的 entry
1336     while (index < objMap->capacity) {
1337         //entries 是个数组, 元素是哈希槽,
1338         //哈希值散布在这些槽中并不连续,因此逐个判断槽位是否在用
1339         if (!VALUE_IS_UNDEFINED(objMap->entries[index].key)) {
1340             RET_NUM(index);       //返回 entry 索引
1341         }
1342         index++;
1343     }
1344
1345     //若没有有效的 entry 了就返回 false,迭代结束
1346     RET_FALSE;
1347 }
1348
1349 //objMap.keyIteratorValue_(_): key=map.keyIteratorValue(iter)
```

仅供非商业用途或交流学习使用


```
1350 static bool primMapKeyIteratorValue(VM* vm, Value* args) {
1351     ObjMap* objMap = VALUE_TO_OBJMAP(args[0]);
1352 
1353     uint32_t index = validateIndex(vm, args[1], objMap->capacity);
1354     if (index == UINT32_MAX) {
1355         return false;
1356     }
1357 
1358     Entry* entry = &objMap->entries[index];
1359     if (VALUE_IS_UNDEFINED(entry->key)) {
1360         SET_ERROR_FALSE(vm, "invalid iterator!");
1361     }
1362 
1363     //返回该 key
1364     RET_VALUE(entry->key);
1365 }
1366 
1367 //objMap.valueIteratorValue_(_):
1368 //value = map.valueIteratorValue_(iter)
1369 static bool primMapValueIteratorValue(VM* vm, Value* args) {
1370     ObjMap* objMap = VALUE_TO_OBJMAP(args[0]);
1371 
1372     uint32_t index = validateIndex(vm, args[1], objMap->capacity);
1373     if (index == UINT32_MAX) {
1374         return false;
1375     }
1376 
1377     Entry* entry = &objMap->entries[index];
1378     if (VALUE_IS_UNDEFINED(entry->key)) {
1379         SET_ERROR_FALSE(vm, "invalid iterator!");
1380     }
1381 
1382     //返回该 key
1383     RET_VALUE(entry->value);
1384 }
…略
1686     //map 类
1687     vm->mapClass = VALUE_TO_CLASS(getCoreClassValue(coreModule, "Map"));
1688     PRIM_METHOD_BIND(vm->mapClass->objHeader.class, "new()", primMapNew);
1689     PRIM_METHOD_BIND(vm->mapClass, "[_]", primMapSubscript);
1690     PRIM_METHOD_BIND(vm->mapClass, "[_]=(_)", primMapSubscriptSetter);
1691     PRIM_METHOD_BIND(vm->mapClass, "addCore_(_,_)", primMapAddCore);
1692     PRIM_METHOD_BIND(vm->mapClass, "clear()", primMapClear);
1693     PRIM_METHOD_BIND(vm->mapClass, "containsKey(_)", primMapContainsKey);
1694     PRIM_METHOD_BIND(vm->mapClass, "count", primMapCount);
1695     PRIM_METHOD_BIND(vm->mapClass, "remove(_)", primMapRemove);
1696     PRIM_METHOD_BIND(vm->mapClass, "iterate_(_)", primMapIterate);
1697     PRIM_METHOD_BIND(vm->mapClass, "keyIteratorValue_(_)", primMapKeyIteratorValue);
1698     PRIM_METHOD_BIND(vm->mapClass, "valueIteratorValue_(_)", primMapValueIteratorValue);
1699 }
```

仅供非商业用途或交流学习使用

函数 primMapRemove 对应于脚本方法 objMap.remove(key)，用于删除 objMap 中的 key。这是通过 removeKey 实现的。

函数 primMapIterate 对应于脚本方法 objMap.iterate_(args[1])，用于迭代 map 中的 entry。开头先获取对象 objMap，判断其 entry 的长度，即 objMap->count 是否为 0，若为 0 说明为空，则直接返回 false 表示迭代结束。如果是第一次迭代，参数 args[1]必为 null，第 1311 行的 index 就为此准备，如果 args[1]不为 null，说明并不是第一次迭代，那么接下来就校验 args[1]必须为正整数，不能越界，随后使 index++更新为下一个索引。接下来的 while 循环利用索引 index 遍历 map 中的 entries 数组，由于哈希值散布在哈希槽中，并不连续，因此需要逐个判断槽位是否在用，如果相应的 key 值并不是 UNDEFINED，就返回 index 作为下一个迭代器。如果没有找到下一个可用的 index 就返回 false。

函数 primMapKeyIteratorValue 对应于方法 objMap.keyIteratorValue_(_)，同样是内部方法，用于返回下一个 key 的迭代器。先获取 objMap 对象，然后调用 validateIndex 获取索引 index，接下来根据 index 获取相应的 entry，若 entry->key 不为 UNDEFINED 说明 key 有效，于是在最后返回 entry->key。

函数 primMapValueIteratorValue 对应于方法 objMap.valueIteratorValue_(_)，同函数 primMapKeyIteratorValue 类似，只是返回的是下一个 value 的迭代器，逻辑相同，不说了。

最后是将脚本方法与原生方法绑定。

8.9 range 类及其方法（本节源码 stepByStep/c8/i）

本节补充 range 类，先看如下的样例代码。

```
[work@work spr]$ cat -n /tmp/test.sp
     1 var r = 2..8
     2 System.print(r.min)
     3 System.print(r.max)
     4 System.print(r.from)
     5 System.print(r.to)
     6 System.print(r.iteratorValue(9999))
     7 System.print(r.iteratorValue(7))
     8
     9 System.print("iterate range")
    10 for i (r) System.print(i)
[work@work spr]$ ./spr /tmp/test.sp
2
8
2
8
false
7
iterate range
2
3
4
5
6
```

仅供非商业用途或交流学习使用


```
7
8
[work@work spr]$
```

下面看如下的实现。

stepByStep/c8/i/vm/core.c

```
...略
1386 //objRange.from: 返回 range 的 from
1387 static bool primRangeFrom(VM* vm UNUSED, Value* args) {
1388      RET_NUM(VALUE_TO_OBJRANGE(args[0])->from);
1389 }
1390
1391 //objRange.to: 返回 range 的 to
1392 static bool primRangeTo(VM* vm UNUSED, Value* args) {
1393      RET_NUM(VALUE_TO_OBJRANGE(args[0])->to);
1394 }
1395
1396 //objRange.min: 返回 range 中 from 和 to 较小的值
1397 static bool primRangeMin(VM* vm UNUSED, Value* args) {
1398      ObjRange* objRange = VALUE_TO_OBJRANGE(args[0]);
1399      RET_NUM(fmin(objRange->from, objRange->to));
1400 }
1401
1402 //objRange.max: 返回 range 中 from 和 to 较大的值
1403 static bool primRangeMax(VM* vm UNUSED, Value* args) {
1404      ObjRange* objRange = VALUE_TO_OBJRANGE(args[0]);
1405      RET_NUM(fmax(objRange->from, objRange->to));
1406 }
1407
1408 //objRange.iterate(_): 迭代 range 中的值,并不索引
1409 static bool primRangeIterate(VM* vm, Value* args) {
1410      ObjRange* objRange = VALUE_TO_OBJRANGE(args[0]);
1411
1412      //若未提供 iter 说明是第一次迭代,因此返回 range->from
1413      if (VALUE_IS_NULL(args[1])) {
1414           RET_NUM(objRange->from);
1415      }
1416
1417      //迭代器必须是数字
1418      if (!validateNum(vm, args[1])) {
1419           return false;
1420      }
1421
1422      //获得迭代器
1423      double iter = VALUE_TO_NUM(args[1]);
1424
1425      //若是正方向
1426      if (objRange->from < objRange->to) {
1427           iter++;
1428           if (iter > objRange->to) {
```

仅供非商业用途或交流学习使用


```
1429                RET_FALSE;
1430            }
1431        } else {  //若是反向迭代
1432            iter--;
1433            if (iter < objRange->to) {
1434                RET_FALSE;
1435            }
1436        }
1437
1438        RET_NUM(iter);
1439 }
1440
1441 //objRange.iteratorValue(_): range 的迭代就是 range 中从 from 到 to 之间的值
1442 //因此直接返回迭代器就是 range 的值
1443 static bool primRangeIteratorValue(VM* vm UNUSED, Value* args) {
1444        ObjRange* objRange = VALUE_TO_OBJRANGE(args[0]);
1445        double value = VALUE_TO_NUM(args[1]);
1446
1447        //确保 args[1]在 from 和 to 的范围中
1448        //若是正方向
1449        if (objRange->from < objRange->to) {
1450            if (value >= objRange->from && value <= objRange->to) {
1451                RET_VALUE(args[1]);
1452            }
1453        } else {  //若是反向迭代
1454            if (value <= objRange->from && value >= objRange->to) {
1455                RET_VALUE(args[1]);
1456            }
1457        }
1458        RET_FALSE;
1459 }
…略
1775        //range 类
1776        vm->rangeClass = VALUE_TO_CLASS(getCoreClassValue(coreModule, "Range"));
1777        PRIM_METHOD_BIND(vm->rangeClass, "from", primRangeFrom);
1778        PRIM_METHOD_BIND(vm->rangeClass, "to", primRangeTo);
1779        PRIM_METHOD_BIND(vm->rangeClass, "min", primRangeMin);
1780        PRIM_METHOD_BIND(vm->rangeClass, "max", primRangeMax);
1781        PRIM_METHOD_BIND(vm->rangeClass, "iterate(_)", primRangeIterate);
1782        PRIM_METHOD_BIND(vm->rangeClass, "iteratorValue(_)", primRangeIteratorValue);
1783 }
```

objRange 就是一段范围，这是用对象中的起始 from 和结束 to 来描述的。

函数 primRangeFrom 对应于脚本方法 objRange.from，用于返回 range 的 from。

函数 primRangeTo 对应于脚本方法 objRange.to，用于返回 range 的 to。

函数 primRangeMin 对应于脚本方法 objRange.min，用于返回 range 中 from 和 to 较小的值。

函数 primRangeMax 对应于脚本方法 primRangeMax，用于返回 range 中 from 和 to 较大的值。

函数 primRangeIterate 对应于脚本方法 objRange.iterate(_)，用于迭代 objRange 中的值。先把

仅供非商业用途或交流学习使用

args[0]转换为 objRange 对象，随后判断 args[1]是否为 null，若是第一次迭代 args[1]为 null，那么返回 from 即可，因为 from 是范围的首元素。下面校验迭代器必须为数字，否则返回 false。验证过后把迭代器 args[1]转换为数字 iter，如果 from 小于 to，即是递增的范围，我们认为是正方向，否则为反方向，下面根据这两类情况分别迭代，最后返回 iter。

函数 primRangeIteratorValue 对应于脚本方法 objRange.iteratorValue(_)，用于迭代 objRange 中的对象，由于 objRange 就是一段范围，并不是可索引的数组，因此这里的迭代就是直接返回范围中的值。如果参数 args[1]属于此范围，那么就直接返回参数 args[1]，否则返回 false。

最后是把脚本方法与原生方法绑定。

8.10 System 类及其方法（本节源码 stepByStep/c8/j）

System 类用于一些和系统相关的功能，比如读取模块、获取系统时间和触发 gc（gc 完成后补充）等，读者看下面的样例代码。

```
[work@work spr]$ cat -n /tmp/testModule.sp
     1 var moduleVar = "var from testModule.sp"
     2 System.print("I am testModule")
[work@work spr]$ cat -n /tmp/test.sp
     1 System.print(System.clock)
     2 System.importModule("testModule")
     3 var testModuleVar = System.getModuleVariable("testModule", "moduleVar")
     4 System.print(testModuleVar)
[work@work spr]$ ./spr /tmp/test.sp
1500343300
I am testModule
var from testModule.sp
[work@work spr]$
```

下面看实现。

stepByStep/c8/j/vm/core.c(1/2)

```
...略
415 //获取文件全路径
416 static char* getFilePath(const char* moduleName) {
417       uint32_t rootDirLength = rootDir == NULL ? 0 : strlen(rootDir);
418       uint32_t nameLength = strlen(moduleName);
419       uint32_t pathLength = rootDirLength + nameLength + strlen(".sp");
420       char* path = (char*)malloc(pathLength + 1);
421
422       if (rootDir != NULL) {
423             memmove(path, rootDir, rootDirLength);
424       }
425
426       memmove(path + rootDirLength, moduleName, nameLength);
427       memmove(path + rootDirLength + nameLength, ".sp", 3);
428       path[pathLength] = '\0';
429
430       return path;
431 }
```

仅供非商业用途或交流学习使用


```
432
433 //读取模块
434 static char* readModule(const char* moduleName) {
435     //1. 读取内建模块，先放着
436
437     //2 .读取自定义模块
438     char* modulePath = getFilePath(moduleName);
439     char* moduleCode = readFile(modulePath);
440     free(modulePath);
441
442     return moduleCode;  //由主调函数将来 free 此空间
443 }
444
445 //输出字符串
446 static void printString(const char* str) {
447     //输出到缓冲区后立即刷新
448     printf("%s", str);
449     fflush(stdout);
450 }
451
452 //导入模块 moduleName,主要是编译模块加载到 vm->allModules
453 static Value importModule(VM* vm, Value moduleName) {
454     //若已经导入则返回 NULL_VAL
455     if (!VALUE_IS_UNDEFINED(mapGet(vm->allModules, moduleName))) {
456         return VT_TO_VALUE(VT_NULL);
457     }
458     ObjString* objString = VALUE_TO_OBJSTR(moduleName);
459     const char* sourceCode = readModule(objString->value.start);
460
461     ObjThread* moduleThread = loadModule(vm, moduleName, sourceCode);
462     return OBJ_TO_VALUE(moduleThread);
463 }
464
465 //在模块 moduleName 中获取模块变量 variableName
466 static Value getModuleVariable(VM* vm, Value moduleName, Value variableName) {
467     //调用本函数前模块已经被加载了
468     ObjModule* objModule = getModule(vm, moduleName);
469     if (objModule == NULL) {
470         ObjString* modName = VALUE_TO_OBJSTR(moduleName);
471
472         //24 是下面 sprintf 中 fmt 中除%s 的字符个数
473         ASSERT(modName->value.length < 512 - 24, "id's buffer not big enough!");
474         char id[512] = {'\0'};
475             int len = sprintf(id, "module \'%s\' is not loaded!", modName->value.start);
476         vm->curThread->errorObj = OBJ_TO_VALUE(newObjString(vm, id, len));
477         return VT_TO_VALUE(VT_NULL);
478     }
479
480     ObjString* varName = VALUE_TO_OBJSTR(variableName);
```

仅供非商业用途或交流学习使用


```
481
482     //从 moduleVarName 中获得待导入的模块变量
483     int index = getIndexFromSymbolTable(&objModule->moduleVarName,
484                 varName->value.start, varName->value.length);
485
486     if (index == -1) {
487         //32 是下面 sprintf 中 fmt 中除%s 的字符个数
488         ASSERT(varName->value.length < 512 - 32, "id's buffer not big enough!");
489         ObjString* modName = VALUE_TO_OBJSTR(moduleName);
490         char id[512] = {'\0'};
491         int len = sprintf(id, "variable \'%s\' is not in module \'%s\'!",
492                 varName->value.start, modName->value.start);
493         vm->curThread->errorObj = OBJ_TO_VALUE(newObjString(vm, id, len));
494         return VT_TO_VALUE(VT_NULL);
495     }
496
497     //直接返回对应的模块变量
498     return objModule->moduleVarValue.datas[index];
499 }
```

函数 getFilePath 用于获取文件绝对路径，文件名在此就是模块名，就是拼路径，不说了。

函数 readModule 用于读取模块，也就是读取文件，在此获取模块文件中的代码，最后返回代码（即字符串）。

函数 printString 用于输出字符串，就是直接调用 printf，然后 fflush 立即刷新缓冲区。

函数 importModule 用于导入模块 moduleName，过程是先编译，然后添加到 vm->allModules 中表示加载完成。开头调用 mapGet 从 vm->allModules 中获取模块 moduleName，判断模块 moduleName 是否已经导入过，若已经导入过则返回 NULL。否则调用 readModule 读取模块，然后调用 loadModule 加载编译模块。

函数 getModuleVariable 用于在模块 moduleName 中获取模块变量 variableName。导入模块变量只能在导入模块之后才能进行，比如已在编译“import 模块 for 模块变量”时生成调用函数 importModule(_) 的指令，也就是已将模块加载到 vm->allModules，但是若通过脚本方法 System. getModuleVariable（“模块名”“模块变量名”）导入模块变量，就等于绕过了导入模块的动作，这是错误的，因此我们就设置 errorObj 对象并返回 null，由虚拟机做下一步处理。也许有读者会说：“为什么不悄悄帮用户导入模块呢？”我个人很不喜欢“不确定性”，比如网页中有很多到达同一页面的链接，看似方便实为模糊，不容易让用户总结出规律，因此我觉得用户知道自己在干什么及应该怎样做，清晰的语法规则对用户掌握一门语言来说很有帮助。如果模块已经导入，那么接下来就获取变量名到 varName，调用 getIndexFromSymbolTable 从模块 moduleVarName 中获得模块变量 varName 的索引 index。如果 index 为-1，说明模块中该模块变量不存在，同样是设置 errorObj 并返回 null。如果模块变量存在，最后就将其返回。

stepByStep/c8/j/vm/core.c(2/2)

```
...略
1590 //System.clock: 返回以秒为单位的系统时钟
1591 static bool primSystemClock(VM* vm UNUSED, Value* args UNUSED) {
1592     RET_NUM((double)time(NULL));
1593 }
```

仅供非商业用途或交流学习使用


```
1594
1595 //System.importModule(_): 导入并编译模块 args[1],把模块挂载到 vm->allModules
1596 static bool primSystemImportModule(VM* vm, Value* args) {
1597     if (!validateString(vm, args[1])) { //模块名为字符串
1598         return false;
1599     }
1600
1601     //导入模块 name 并编译 , 把模块挂载到 vm->allModules
1602     Value result = importModule(vm, args[1]);
1603
1604     //若已经导入过则返回 NULL_VAL
1605     if (VALUE_IS_NULL(result)) {
1606         RET_NULL;
1607     }
1608
1609     //若编译过程中出了问题,切换到下一线程
1610     if (!VALUE_IS_NULL(vm->curThread->errorObj)) {
1611         return false;
1612     }
1613
1614     //回收 1 个 slot 空间
1615     vm->curThread->esp--;
1616
1617     ObjThread* nextThread = VALUE_TO_OBJTHREAD(result);
1618     nextThread->caller = vm->curThread;
1619     vm->curThread = nextThread;
1620     //返回 false,vm 会切换到此新加载模块的线程
1621     return false;
1622 }
1623
1624 //System.getModuleVariable(_,_): 获取模块 args[1]中的模块变量 args[2]
1625 static bool primSystemGetModuleVariable(VM* vm, Value* args) {
1626     if (!validateString(vm, args[1])) {
1627         return false;
1628     }
1629
1630     if (!validateString(vm, args[2])) {
1631         return false;
1632     }
1633
1634     Value result = getModuleVariable(vm, args[1], args[2]);
1635     if (VALUE_IS_NULL(result)) {
1636         //出错了,给 vm 返回 false 以切换线程
1637         return false;
1638     }
1639
1640     RET_VALUE(result);
1641 }
1642
```

仅供非商业用途或交流学习使用


```
1643 //System.writeString_(_): 输出字符串 args[1]
1644 static bool primSystemWriteString(VM* vm UNUSED, Value* args) {
1645       ObjString* objString = VALUE_TO_OBJSTR(args[1]);
1646       ASSERT(objString->value.start[objString->value.length] == '\0', "string
          isn't terminated!");
1647       printString(objString->value.start);
1648       RET_VALUE(args[1]);
1649 }
…略
1930       //system 类
1931       Class* systemClass = VALUE_TO_CLASS(getCoreClassValue(coreModule, "System"));
1932       PRIM_METHOD_BIND(systemClass->objHeader.class, "clock", primSystemClock);
1933       PRIM_METHOD_BIND(systemClass->objHeader.class, "importModule(_)",
          primSystemImportModule);
1934       PRIM_METHOD_BIND(systemClass->objHeader.class, "getModuleVariable(_,_)",
          primSystemGetModuleVariable);
1935       PRIM_METHOD_BIND(systemClass->objHeader.class, "writeString_(_)",
          primSystemWriteString);
1936 }
```

函数 primSystemClock 对应于脚本方法 System.clock，用于返回秒为单位的系统时钟，这是直接调用 time 函数实现的，返回值是 1970 年 1 月 1 日 00:00:00（称为 UNIX 系统的 Epoch 时间）到当前时刻的秒数。

函数 primSystemImportModule 对应于 System.importModule(args[1])，功能是导入并编译模块 args[1]，把模块挂载到 vm->allModules。我们在“import 模块名 for 模块变量名”结构中，对模块名和模块变量名不要求加入双引号，而在 System.importModule（“模块名”）中，模块名必须要用双引号括起来以表示是字符串，否则模块名就变成变量名了，除非变量的值就是字符串。比如 var mdName = "testModule"，然后这样调用 System.importModule(mdName)也行。接下来调用函数 importModule 导入模块并编译，最后会挂载到 vm->allModules 之下表示已经导入。正常情况下 importModule 的返回值是运行该模块的新线程的 Value 结构，如果返回值是 null 的 Value 结构，说明该模块之前已经导入过了，那么本函数第 1606 行直接返回 null。如果导入过程中出错，会在 curThread->errorObj 中设置报错信息，这种情况则返回 false，告诉虚拟机切换线程。为调用 System.importModule(moduleName)，已经在主调方的栈中压入了 system 和 moduleName（即 args[]1），这里只需要一个 slot 来存储返回值，因此第 1615 行回收一个空间（args[0]的空间最终有返回值的接收者即函数的主调方回收）。最后将 result 还原为线程 nextThread 准备运行，将其主调方即 nextThread->caller 设置为 nextThread，然后使当前运行的线程指针 vm->curThread 设置为 nextThread，最后返回 false，告诉虚拟机运行 nextThread。

函数 primSystemGetModuleVariable 对应于脚本方法 System.getModuleVariable(_,_)，args[1]是模块名，args[2]是变量名，本函数用于获取模块 args[1]中的模块变量 args[2]。函数开头校验模块名和变量名的类型必须为字符串。然后调用 getModuleVariable 获取模块变量，最后返回模块变量 result。

函数 primSystemWriteString 对应于脚本方法 System.writeString_(_)，用于输出字符串 args[1]，内部使用。函数是直接调用 printString 完成的输出。

最后是把脚本方法和原生绑定。

8.11 收尾与测试（本节源码 stepByStep/c8/k）

本节是一些收尾工作，完成之后我们便测试解释器的功能。

stepByStep/c8/k/vm/core.c

```
…略
1734 //编译核心模块
1735 void buildCore(VM* vm) {
…略
1935     PRIM_METHOD_BIND(systemClass->objHeader.class, "writeString_(_)", primSystemWriteString);
1936
1937     //在核心自举过程中创建了很多 ObjString 对象,创建过程中需要调用 initObjHeader 初始化对象头
1938     //使其 class 指向 vm->stringClass，但那时的 vm->stringClass 尚未初始化,因此现在更正
1939     ObjHeader* objHeader = vm->allObjects;
1940     while (objHeader != NULL) {
1941       if (objHeader->type == OT_STRING) {
1942           objHeader->class = vm->stringClass;
1943       }
1944       objHeader = objHeader->next;
1945     }
1946 }
```

在核心自举过程中创建了很多 ObjString 对象，创建过程中需要调用 initObjHeader 初始化对象头，使对象头中的 class 成员指向 vm->stringClass，但那时的 vm->stringClass 尚未初始化，即那时类 String 未挂载到 vm->stringClass 中，比如在函数 buildCore 尚未为 vm->stringClass 类赋值之前，又在 buildCore 中调用 loadModule，进而调用 loadModule->compileModule->compileProgram->compileClassDefinition->newObjString->initObjHeader，函数 initObjHeader 中将类头中的 class 成员指向 vm->stringClass，而 vm->stringClass 那时尚未赋值，这将导致引用字符串方法时出错，而现在已经把类 String 挂载到 vm->stringClass 下了，所以现在需要修正那些指向 vm->stringClass 的类。代码第 1939～1945 行，通过遍历所有对象链表 vm->allObjects，逐个找出字符串类型，使其指向已经初始化的 vm->stringClass。

下面需要在 cli.c 中把代码 printf("There is something to do...\n"); exit(0);去掉，这样就能够编译用户的脚本文件了，修改后的 cli.c 如下所示。

stepByStep/c8/k/cli/cli.c

```
…略
 8 //执行脚本文件
 9 static void runFile(const char* path) {
10     const char* lastSlash = strrchr(path, '/');
11     if (lastSlash != NULL) {
12         char* root = (char*)malloc(lastSlash - path + 2);
13         memcpy(root, path, lastSlash - path + 1);
14         root[lastSlash - path + 1] = '\0';
15         rootDir = root;
16     }
17
18     VM* vm = newVM();
```

```
 19      const char* sourceCode = readFile(path);
 20      executeModule(vm, OBJ_TO_VALUE(newObjString(vm, path, strlen(path))), sourceCode);
 21 }
…略
```

一切就绪，现在可以试试编译运行下面的脚本，这是本书开头的例子，方便大家就再次贴过来。

```
[work@work k]$ cat -n employee.sp
     1 class Employee {
     2     var name
     3     var gender
     4     var age
     5     var salary
     6     static var employeeNum = 0
     7     new(n, g, a, s) {
     8        name = n
     9        gender = g
    10        age = a
    11        salary = s
    12        employeeNum = employeeNum + 1
    13     }
    14
    15     sayHi() {
    16        System.print("My name is " +
    17         name + ", I am a " + gender +
    18         ", " + age.toString + "years old")
    19     }
    20
    21     salary {
    22         return salary
    23     }
    24
    25     static employeeNum {
    26         return employeeNum
    27     }
    28 }
[work@work k]$ cat -n manager.sp
     1 /*
     2     本文件中的代码只为演示语法,无任何意义,不用深究
     3     精力有限,这里只演示部分功能,如果读者感兴趣,
     4     可以参考 core.c 中后面注册的原生方法和 core.script.inc 中的脚本方法
     5     测试有限,难免还会有 bug,读过本书后应该有 bugfix 的能力，看好你,兄弟
     6                         刚子
     7                         2018.4.12
     8 */
     9
    10 import employee for Employee
    11 var xh =  Employee.new("xiaohong", "female", 20, 6000)
    12 System.print(xh.salary)
    13
    14 var xm =  Employee.new("xiaoming", "male", 23, 8000)
```

```
15 System.print(xm.salary)
16
17 System.print(Employee.employeeNum)
18
19 class Manager < Employee {
20     var bonus
21     bonus=(v) {
22         bonus = v
23     }
24
25     new(n, g, a, s, b) {
26         super(n, g, a, s)
27         bonus = b
28     }
29
30     salary {
31         return super.salary + bonus
32     }
33
34 }
35
36 fun employeeInfo() {
37     System.print("number of employee:" + Employee.employeeNum.toString)
38     var employeeTitle = Map.new()
39     employeeTitle["xh"] = "rd"
40     employeeTitle["xm"] = "op"
41     employeeTitle["lw"] = "manager"
42     employeeTitle["lz"] = "pm"
43
44     for k (employeeTitle.keys) {
45         System.print(k + " -> " + employeeTitle[k])
46     }
47
48     var employeeHeight = {
49         "xh": 170,
50         "xm": 172,
51         "lw": 168,
52         "lz": 173
53     }
54     var totalHeight = 0
55     for v (employeeHeight.values) {
56         totalHeight = totalHeight + v
57     }
58     System.print("averageHeight:%(totalHeight/ employeeHeight.count)")
59
60     var allEmployee = ["xh", "xm", "lw", "lz"]
61     for e (allEmployee) {
62         System.print(e)
63     }
64
```

```
    allEmployee.add("xl")
    System.print("all employee are:%(allEmployee.toString)")
    var idx = 0
    var count = allEmployee.count
    while (idx < count) {
       System.print(allEmployee[idx])
       idx = idx + 1
    }

    //System.gc()  //可以手动回收内存

    var a = 3 + 5 > 9 - 3  ? "yes" : "no"
    if (a.endsWith("s")) {
        System.print(System.clock)
    } else {
        System.print("error!!!!!")
    }

    var str = "hello, world."
    System.print(str[-1..0])
}

var lw = Manager.new("laowang", "male", 35, 13000, 2000)
System.print(lw.salary)
lw.bonus=3100
System.print(lw.salary)
var lz = Manager.new("laozheng", "male", 36, 15000, 2300)
System.print(lz.salary)

var thread = Thread.new(employeeInfo)
thread.call()
```

```
[work@work k]$ ./spr manager.sp
6000
8000
2
15000
16100
17300
number of employee:4
xm -> op
lz -> pm
lw -> manager
xh -> rd
averageHeight: 170.75
xh
xm
lw
lz
all employee are:[xh,xm,lw,lz,xl]
xh
```

```
xm
lw
lz
xl
1501988169
.dlrow ,olleh
[work@work k]$
```

由于此时我们尚未完成垃圾回收，所以注掉了 System.gc()。

第9章 垃圾回收

9.1 垃圾回收浅述

垃圾回收即我们常说的 Garbage Collection，简称 GC。

首先搞清楚两件事：什么是垃圾（Garbage）以及为什么要把垃圾回收（Collection）。“垃圾”的概念大家都明白，就是指那些没有用、已经不需要的东西。比如生活上的垃圾有饮料瓶、包装袋等，工业上的垃圾有原材料的下脚料、废油等。那么“回收”是在做什么？从字面上看垃圾回收的意思似乎是：把垃圾回收，为了环保而二次利用这些垃圾，或者可循环利用垃圾的材料而制造出新产品，比如啤酒瓶可直接被啤酒厂回收，清洗干净后可再次装啤酒。以上是“垃圾回收”在生活中的意义，即二次利用这些“垃圾本身”。而在计算机中的垃圾回收并不是这个意思。计算机中的垃圾指的是那些没用的数据，既然没用了，那么回收它们也无法二次利用，因此回收的目的并不是为了要垃圾数据本身，而是为了腾出这些垃圾数据所占的“内存空间”，然后把该“内存空间”给新的数据使用。重复一下，这里强调的是“空间”，即垃圾回收在计算机中的意义是回收垃圾数据所在的内存空间，是内存空间有用，并不是垃圾数据本身有用，垃圾数据本身就丢弃了。

以下对垃圾回收简称 GC。

GC 是很早就有的概念，最初的 GC 算法是由 John McCarthy 在 1960 年发布的，John McCarthy 是一位非常了不起的黑客，他不仅是 Lisp 语言之父，还是人工智能之父，来头好大，然而让 GC 一举成名的却是 Java 语言，因为 Java 应用太广泛了。

使用 GC 有什么好处呢？比如程序员在 C 或 C++语言中用 malloc 和 new 函数从堆中申请的内存空间，需要调用 free 或 delete 函数将其释放，即空间被操作系统的内存管理机制回收。无论程序员申请了多少个内存空间，每申请一次就要对应着释放一次，否则光申请不释放，那么程序在运行时所占的空间会越来越多，这会造成所谓的“内存泄露”。这其实是一件体力活儿，必须由程序员主动释放所申请的内存，也就是程序员得记着在哪里有过内存申请，然后在使用过后及时释放该内存。可是在实际使用中往往情况更复杂，也许程序员为了追求效率，所申请的内存可能在别处另有它用，因此并不会在使用过后马上“就地”释放，而是在“别处”用完后再释放，这就容易导致忘记释放的问题。当程序出现内存泄漏时程序员就应该及时修复，否则内存耗尽会影响到业务。可是如果程序员一时无法找到哪里的内存忘记释放了呢？这很常见，通常程序员为查一个内存泄漏，如果程序复杂的话，往往要耗上一周以上的时间。对于这种情况只有通过杀掉进程来解决了（当然这也只是临时的方法，下次内存吃紧时还要杀掉该进程），因为进程退出（被杀掉也是退出）时，操作系统会将进程所申请的内存空间全部回收。

程序员忘记释放的内存，操作系统的内存管理系统可不会忘记，在程序退出相关的系统调用中（如_exit()）会触发内存管理系统的内存回收函数，该函数会遍历该进程的页表回收进程所占

据的所有内存。说明一下，因为进程使用的任何内存都会记录在页表中，包括进程自身的静态内存和从堆中分配的动态内存。在页表中的记录的内存块叫“页”，也叫页框，页大小通常是以 4K 的倍数为单位，后面所说的页的换入换出，就是把这页大小的内存块在物理内存和硬盘之间挪来挪去。

另外，由于申请的内存往往用一个指针来记录，在释放了内存（由该指针指向）后，该指针依然指向已释放的内存，由于该内存地址已无效了，这种指针便称为悬垂指针（dangling pointer），相当于指针是被悬挂起来的，无处安放，因为指针指向的地址并不是其该指向的地方，因此是无效的、假的、虚的。由于指针指向的地址是不可预测的，因此如果引用该悬垂指针就会产生不可预测的错误。

无论是脚本程序还是二进制可执行程序，只分配内存而不释放的话，程序所占的空间必然会越来越大，如果不加控制，一定会耗尽物理内存，当然大多数操作系统都支持虚拟内存管理，直接耗尽的是物理内存，而间接耗尽的是虚拟内存。这里所说的虚拟内存管理和进程的虚拟地址空间是两回事。在没有虚拟地址空间以前，计算机运行在“实”模式下，“实”的意义是，进程访问的地址和物理内存是真“实”对应的，即程序若访问第 1 字节，那么就是访问物理内存的第 1 字节，这种一一对应的访问是缺乏安全性的，因为这会让进程访问到操作系统或其他核心数据的空间，比如地址 0～0x3FF 是中断向量表的空间，用户进程完全可以改写这段内存中的数据，熟悉微软操作系统 Windows 98 的读者肯定熟悉，运行某个程序时 Windows 98 上经常会提示“非法访问”，然后程序就退出了，本质上就是这种情况。提示一下，Windows 98 只是 DOS 的一个图形界面，本身还是实模式 DOS 系统，图形上的鼠标点击操作都被转换为命令行来实现。后来为了提升安全性和地址空间等，计算机硬件上产生了后来的保护模式，即安全性得到了保护，而虚拟地址空间便是保护模式下的产物。虚拟地址空间是属于进程的，是进程可使用的内存地址范围，是逻辑上的，并不受限于物理内存大小，比如 32 位系统下进程的虚拟地址空间是 4GB，尽管物理内存没那么大，但进程却拥有了比物理内存更广阔的 4GB 地址空间，比如物理内存是 1GB，那么访问第 2GB 字节时也没问题，背后原理是通过进程页表中虚拟地址到物理地址的映射转换来实现的。而在背后支持这种映射转换的，就是操作系统的虚拟内存管理机制，切记，虚拟内存管理机制是属于操作系统的范畴。虚拟地址只是虚拟的，最终也是作用到某个物理地址空间上，因此间接也会受物理内存大小的影响，但这影响只是体现在虚拟地址到物理地址映射的“效率”上，程序运行是不受限的，原因是一般情况下物理内存空间远小于进程的虚拟地址空间，这才是虚拟内存地址空间提出的意义（也许有读者会说，我机器装了 4GB 内存呢，和虚拟地址空间一样大，用不着虚拟地址了吧？估计你肯定说的是 32 位系统下，如果是 64 位系统估计这位读者就没那么“嚣张”了，64 位系统支持的虚拟地址空间是 2 的六十四次方，即 16777216TB，也就是 17179869184GB，如果也装这么大的物理内存，我想这可不是任何人都装得起的，甚至我怀疑当今的主板是否支持这么大的内存容量）。而硬盘空间往往更大（但也没大到 16777216TB，这里只讨论一般情况），因此进程若申请了超过物理内存大小的内存空间（进程本身静态数据很大或进程运行过程中申请了很多内存），为了使进程正常运行，操作系统要把硬盘空间临时借用一下了，把物理内存中的该进程暂时不用的内存（页）换出到硬盘，即复制到硬盘上，这就在物理内存上腾出了一部分空间，然后再从页表中将虚拟地址映射到该空间，这样该进程就可以使用相应的虚拟地址了。

操作系统是如何知道虚拟地址对应的内存（页）不在物理内存中的呢？这就需要 CPU 硬件的支持了，可以通过缺页异常（Page Fault）来实现，这是 CPU 硬件原生支持的功能，当某个内存页不在内存中时，CPU 会触发缺页异常，操作系统为缺页异常提供了一个回调函数，因此缺页时会调用该函数，该函数把某不常用的内存地址 A 中的数据换到硬盘上存储，然后把引起缺页中断的内存地址 B 中的数据（已由之前的缺页中断存储到硬盘中）载入到物理内存使程序可以正常运

行。然后在需要内存地址 A 时再把硬盘中内存地址 A 对应的数据换回到内存中。对于 32 位系统来说，通过虚拟内存管理机制，只要硬盘大于 4GB，即使物理内存只有两个页大小（一个页运行操作系统，一个页运行用户程序），程序也能跑起来。即使是这样，程序的虚拟内存空间毕竟是有限的，如果无限地申请内存，进程的虚拟地址耗尽了，进程就彻底无法运行了。

我们在编写 Java、PHP、Ruby、Perl 或 Python 代码时就不要求显式释放内存，这就是 GC 的功劳，它在背后悄悄为我们回收了那些垃圾所占的内存空间，然后将省出来的内存重新利用。有读者可能不清楚 GC 做了哪些事，那就用 Python 举个具体的例子，在某局部作用域中调用内部函数 list()生成了一个列表对象 li，对象 li 只属于该作用域，故对象 li 是局部对象。出该作用域之前，程序员并没有显式调用 list_free(li)（此方法是我杜撰的，Python 没必要提供，因为有 GC，list_free(li) 在此相当于 C++中的 delete li 之类的）之类的方法去释放对象 li，出了该作用域后，对象 li 的生存期就过了，那么对象 li 的数据就没用了，数据所在的空间就要被回收，而 GC 就（定期或在某条件下触发）在幕后悄悄回收这些“过期”的对象——即垃圾对象。而在 C 或 C++中，每 new 一个对象就必须要用 delete 去释放，否则就内存泄漏，这就是有或没有 GC 的差别。

9.2 理论基础

关于堆的概念，如图 9-1 所示。

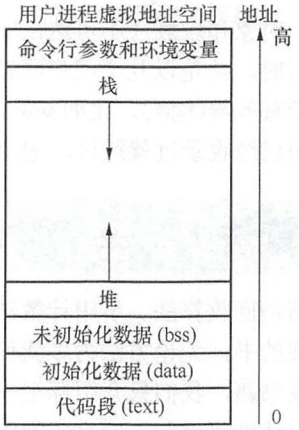


图 9-1

堆是用户进程虚拟地址空间中位于程序本身数据（数据段和代码段）所在空间之上的一段内存，但此段空间是和栈共享的，堆的地址发展方向是由低到高，即在不释放内存的情况下，每次申请内存时，系统给返回的地址是越来越高。而栈的地址发展方向是由高到低，此地址是指栈顶 esp 的值，即 CPU 执行指令 push 时，会把 esp 的值减一个数据单位大小，即 esp 越来越低。因此堆是向上发展，栈是向下发展，这可能会导致碰撞，也就是图 9-1 中两个箭头碰到一起了，再不阻止的话，双方都会进入对方的内存区域，从而破坏对方的数据，因此这需要操作系统有相应的碰撞检测方法。说到此处大家需要明确：因为申请的内存都是从堆中申请的，而对象也是通过申请内存的方式构建的，因此对象回收指的是回收在堆中创建的对象。

赋值器和回收器：对于使用垃圾回收算法的程序即解释器，其执行过程被划分为两个独立的部分——赋值器和回收器。赋值器是指产生垃圾的程序代码，垃圾是指无用的对象，不过它虽然无用但也是对象，在成为垃圾之前肯定是有用的，至少曾经“有用过”。而对象是由解释器申请内存后构造出来的，因此赋值器就是指解释器中用于分配对象的那些代码。回收器是指解释器中用于回收

那些由赋值器生成的对象中的垃圾对象的代码，垃圾对象用术语来说就是下面介绍的不可达对象。

对象的存活性：对象创建后，在解释器后续运行中依然可被访问，也就是会再次用到该对象，那么该对象就是存活的，具有存活性。一般来说，会把具有存活性的对象称为活动对象。

回收器的正确性：回收器保证永远不会回收那些存活的对象，这样的回收器具有正确性。

对象的可达性：一般情况下无法确认对象的存活性，因此无法确认解释器后续运行中是否还会访问该对象，但我们可以用对象的“可达性”近似表示这种存活性——如果从根对象通过指针链可访问到某对象，则称该对象可达，该对象具有可达性。

赋值器的根：根就是 root，即起点，在 GC 中，所有在堆中分配的“可达”对象会以某种方式组织到一起，要想遍历这些对象，总该有个遍历的入口，根就是这些“可达”对象的起始指针，通过根就可以遍历到所有“可达”对象。根既然是遍历的入口，就是可（由赋值器）直接引用的对象，因此根也称为根对象。强调一下，根对象就是回收器进行递归回收工作的入口。由于很多逻辑单元之间彼此是独立的，各逻辑单元内部的对象之间并无关联，要想遍历逻辑单元中的所有对象，也要通过这个逻辑单元的根对象，一个逻辑单元中只能遍历到该逻辑单元内的所有对象，因此根对象就不只一个，有多少个独立的逻辑单元就有多少个根对象。说明一下，逻辑单元只是对象从耦合度上的划分，也就是根据对象所属、依附关系划分出的多个不相交的对象集合。一个或多个根对象组成的集合叫根集合，回收器会遍历集合中各个根对象，对各个根对象采取深度优先的方式处理其内部的对象，这样就处理了该逻辑单元中的所有对象。但是读者不要搞混了，猛地一看这个“根的描述”很像我们的 VM->allObjects，VM->allObjects 是“所有已分配对象链表”的头节点，即包括所有活动（可达）对象和垃圾（不可达，非存活）对象，因此并不能称为根对象。就像搜索引擎的 spider 收录网页时，总是以几个门户网站作为递归的入口（并不是胡乱猜测网站域名，然后瞎撞，有就收录，没有就继续猜），比如 www.ptpress.com.cn，递归收录页面中所链接的页面，如果该链接对应的页面已经收录过就跳过，避免无限递归。

9.3 标记—清扫回收算法

常用的垃圾回收算法有标记—清扫回收算法、引用计数法和 GC 复制法，还有一些从这 3 种方法衍生出来的算法。这是一本实战的书，无论方法再多我们也只是使用其中的一种，因此理论部分就不多介绍了，大家可参考专业书籍，我们只介绍标记—清扫回收算法。

标记—清扫回收算法是在对象可达性“递归”定义下最简单的垃圾回收方案，这里强调了递归，递归是此算法的精髓（好像很多高级的算法都离不开递归），后面会看到我们在实现中也会递归标记活动对象。正如字面意思一样，标记—清扫回收算法在处理过程中分为标记和清扫两个阶段，标记阶段其实就是追踪所有活动对象的阶段，回收器从根集合开始遍历并标记所有活动对象，以表示要保留它们，不能回收。注意，这里说的是标记那些活动对象，就是说明已经知道哪些对象是存活的，也许有读者觉得很晕，心想，我怎么知道哪些是存活的？其实这很简单，编译器是我们设计的，我们当然知道哪些对象要保留了，比如内部对象和没出作用域生存期的对象肯定是要保留的，后面在实现中就明白了。在清扫阶段中，回收器检查每个对象是否做了标记，并将未标记的对象认为是垃圾对象，然后将垃圾对象全部回收，从而达到了回收内存空间的目的。说白了，标记—清扫回收算法是一种间接回收的算法，它并不是直接标记哪些对象是垃圾，而是标记哪些对象要保留（不是垃圾），其余没标记的都是应被回收的垃圾对象。

“三色抽象法”是一种描述追踪式回收器很有效的方法，可用来描述对象在回收器运行的过程中的状态变化。注意，此时说的对象是所有对象，包括垃圾对象和存活对象（保留对象）。回收器在处理过程中，对象会处于不同的状态，比如处于“被打上标记的状态”、处于“处理中”的状态

等。回收器把存活的对象标记为黑色，非存活的对象（垃圾对象）标记为白色。初始状态下默认所有对象都是垃圾对象，因此默认所有对象均为白色。回收器开始遍历所有对象时，将正在处理的对象置为灰色，即白色和黑色的中间态，当处理完该对象后，回收器将其置为黑色以表示保留该对象。因为标记—清扫回收算法先要确定哪些对象是保留的，因此，所有对象的初始颜色是白色，回收器将部分对象标记为灰色以表示处理中，最后再将灰色对象标记为黑色以表示保留、不要回收。对象颜色的变化是白->灰->黑。那些完全没处理过的白色对象就被认为是垃圾对象。灰色表示“处理中”的对象，也许还需进一步的处理，总之未处理完，因此，处理过程结束之后，所有对象的颜色要么是黑，要么是白，肯定没有灰色对象。

标记—清扫回收算法是比较简单的实现，以上理论已经够用了，更多的理论知识请参考专业资料。

9.4 一些基础结构（本节源码 stepByStep/c9/a）

在实现 GC 相关代码之前，有一些依赖的结构要提前准备好，主要是 vm.h 和 vm.c 中，见如下代码。

stepByStep/c9/a/vm/vm.h

```
...略
12 //最多临时根对象数量
13 #define MAX_TEMP_ROOTS_NUM 8
...略
20 typedef enum vmResult {
21     VM_RESULT_SUCCESS,
22     VM_RESULT_ERROR
23 } VMResult;    //虚拟机执行结果
24 //如果执行无误,可以将字符码输出到文件缓存,避免下次重新编译
25
26 //灰色对象信息结构
27 typedef struct {
28     //gc 中的灰对象(也是保留对象)指针数组
29     ObjHeader** grayObjects;
30     uint32_t capacity;
31     uint32_t count;
32 } Gray;
33
34 typedef struct {
35     //堆生长因子
36     int heapGrowthFactor;
37
38     //初始堆大小,默认为 10MB
39     uint32_t initialHeapSize;
40
41     //最小堆大小,默认为 1MB
42     uint32_t minHeapSize;
43
44     //第一次触发 gc 的堆大小,默认为 initialHeapSize
45     uint32_t nextGC;
```

```
46 } Configuration;
47
48 struct vm {
49     Class* classOfClass;
50     Class* objectClass;
51     Class* stringClass;
52     Class* mapClass;
53     Class* rangeClass;
54     Class* listClass;
55     Class* nullClass;
56     Class* boolClass;
57     Class* numClass;
58     Class* fnClass;
59     Class* threadClass;
60
61     uint32_t allocatedBytes;  //累计已分配的内存量
62     ObjHeader* allObjects;  //所有已分配对象链表
63     SymbolTable allMethodNames;     //(所有)类的方法名
64     ObjMap* allModules;
65     ObjThread* curThread;   //当前正在执行的线程
66     Parser* curParser;  //当前词法分析器
67
68     //临时的根对象集合(数组),存储临时需要被 GC 保留的对象,避免回收
69     ObjHeader* tmpRoots[MAX_TEMP_ROOTS_NUM];
70     uint32_t tmpRootNum;
71
72     //用于存储存活(保留)对象
73     Gray grays;
74     Configuration config;
75 };
…略
```

GC 是在任意时刻运行的，有了 GC 之后，解释器在执行过程中创建的对象也有可能被当成垃圾回收，这些对象的显著特点是：“目前只是构建对象阶段，尚未派上用场，因此不应该被回收，以后用过后可以回收”。这种需求是合理的，肯定不能回收过程中的产物，因此我们需要为这些“不可达但不希望被回收”的对象发放“免死金牌”，我们为其创建个“白名单”，并且和 GC 约定：位于白名单中的对象不会被回收。我们是以根对象的方式临时保护这些对象，把这些“免死”的对象放到一个叫 tmpRoots，即“临时根”的数组中，后面会看到这个数组。那么此数组多大呢？这就是第 13 行宏 MAX_TEMP_ROOTS_NUM 的作用，其值为 8，最多同时支持 8 个临时的根对象。现在你不清楚没关系，等介绍完整个 GC 的代码，在了解到工作流程之后就明白了。

标记—清扫回收算法就是从那些默认为垃圾的对象中“挑出”哪些不是垃圾，整个“挑”的过程是用颜色来标识（或者说是跟踪）的，被挑出的对象会有不同的颜色变化，对象颜色的变化是白色→灰色→黑色，即默认为所有对象均为不可达的白色，即所有对象都是垃圾，然后从中“挑”出要保留的对象，但是“挑”也要有个过程，也就是回收器处理过程，我们把位于处理过程中的对象标记为灰色以表示该对象正在被处理，最后再将对象标黑以表示确认其可以保留。

第 27～32 行的 Gray 结构用于收集所有灰色对象，成员 grayObjects 是个指针数组，数组元素便是灰色对象的地址。成员 capacity 和 count 分别表示此数组的容量及实际使用的个数。

结构 Configuration 用来存储虚拟机的配置信息，注释很清楚，不说了。

在 vm 结构中，我们添加了成员 tmpRoots，它表示临时根对象集合，前面刚提到过它，此数组是那些“不可达但又不想被回收的对象”的白名单，一会儿在 vm.c 中会介绍其用法。成员 tmpRootNum 是数组中实际使用的元素个数，因为是从前往后写数组，因此 tmpRootNum 也起到写入指针的作用，用来跟踪数组 tmpRoots 中当前可写入数据的元素索引。成员 grays 和 config 分别用来管理灰对象和配置信息。

stepByStep/c9/a/vm/vm.c

```
...略
//把 obj 作为临时的根对象,就是把 obj 添加为 gc 的白名单,避免被 gc 回收
void pushTmpRoot(VM* vm, ObjHeader* obj) {
    ASSERT(obj != NULL, "root obj is null!");
     ASSERT(vm->tmpRootNum < MAX_TEMP_ROOTS_NUM, "temporary roots too much!");
    vm->tmpRoots[vm->tmpRootNum++] = obj;
}

//去掉临时的根对象
void popTmpRoot(VM* vm) {
      ASSERT(vm->tmpRootNum < MAX_TEMP_ROOTS_NUM, "temporary roots too much!");
    vm->tmpRootNum--;
}

//初始化虚拟机
void initVM(VM* vm) {
    vm->allocatedBytes = 0;
    vm->allObjects = NULL;
    vm->curParser = NULL;
    StringBufferInit(&vm->allMethodNames);
    vm->allModules = newObjMap(vm);
    vm->curParser = NULL;
    vm->config.heapGrowthFactor = 1.5;

    //最小堆大小为 1MB
    vm->config.minHeapSize = 1024 * 1024;
    //初始堆大小为 10MB
    vm->config.initialHeapSize = 1024 * 1024 * 10;

    vm->config.nextGC = vm->config.initialHeapSize;
    vm->grays.count = 0;
    vm->grays.capacity = 32;

    //初始化指针数组 grayObjects
    vm->grays.grayObjects =
       (ObjHeader**)malloc(vm->grays.capacity * sizeof(ObjHeader*));
}
...略
```

函数 pushTmpRoot 把参数 obj 作为临时的根对象，就是把 obj 添加为 gc 的白名单，避免被 gc 回收，实际上就是写数组 vm->tmpRoots 并同时维护写入指针 vm->tmpRootNum，往数组尾部追回

待保留的对象，并使 vm->tmpRootNum 自加 1 以更新下一个写入的位置。

函数 popTmpRoot 和 pushTmpRoot 功能相反，从数组 vm->tmpRoots 中去掉最后一个对象，即使其脱离庇护，参与 GC 的回收。

接下来在 initVM 中添加相关配置的初始化代码，大家自己看下，不说了。

9.5 实现 GC（本节源码 stepByStep/c9/a）

在标记—清扫回收算法中，被保留的对象的颜色是从白色变为灰色，最后变为黑色。但这只是逻辑流程，实现中未必要显式置为白色、灰色或黑色，就像从一堆苹果中把好的捡出来放一堆儿，没必要再多此一举把剩下的苹果贴上“这些是烂苹果”的标签，因为这些苹果不是好的就是烂的。GC 中也一样，回收器回收的是垃圾对象，回收是在清扫阶段进行的，而其前一阶段即标记阶段完成后，对象要么是白色，要么是黑色，回收器回收那些非黑色（即白色）对象就行了，因此我们只用对象头中的 bool 成员 isDark 来表示该对象是黑色，如果 isDark 为 true 则其所在的对象为黑色，否则为白色。

下面介绍 GC 的实现，这次建立目录 gc，在 GC 中包括 gc.c 及 gc.h（头文件中只是一些函数声明，不再列出）。

stepByStep/c9/a/gc/gc.c (1/5)

```
#include "gc.h"
#include "compiler.h"
#include "obj_list.h"
#include "obj_range.h"
#if DEBUG
    #include "debug.h"
    #include <time.h>
#endif

//标灰 obj:即把 obj 收集到数组 vm->grays.grayObjects
void grayObject(VM* vm, ObjHeader* obj) {
    //如果 isDark 为 true 表示为黑色,说明已经可达,直接返回
    if (obj == NULL || obj->isDark) return;

    //标记为可达
    obj->isDark = true;

    //若超过了容量就扩容
    if (vm->grays.count >= vm->grays.capacity) {
        vm->grays.capacity = vm->grays.count * 2;
        vm->grays.grayObjects =
          (ObjHeader**)realloc(vm->grays.grayObjects, vm->grays.capacity *
           sizeof(ObjHeader*));
    }

    //把 obj 添加到数组 grayObjects
    vm->grays.grayObjects[vm->grays.count++] = obj;
}

```

```
29 //标灰 value
30 void grayValue(VM* vm, Value value) {
31     //只有对象才需要标记
32     if (!VALUE_IS_OBJ(value)) {
33         return;
34     }
35     grayObject(vm, VALUE_TO_OBJ(value));
36 }
37
38 //标灰 buffer->datas 中的 value
39 static void grayBuffer(VM* vm, ValueBuffer* buffer) {
40     uint32_t idx = 0;
41     while (idx < buffer->count) {
42         grayValue(vm, buffer->datas[idx]);
43         idx++;
44     }
45 }
46
47 //标黑 class
48 static void blackClass(VM* vm, Class* class) {
49     //标灰 meta 类
50     grayObject(vm, (ObjHeader*)class->objHeader.class);
51
52     //标灰父类
53     grayObject(vm, (ObjHeader*)class->superClass);
54
55     //标灰方法
56     uint32_t idx = 0;
57     while (idx < class->methods.count) {
58         if (class->methods.datas[idx].type == MT_SCRIPT) {
59             grayObject(vm, (ObjHeader*)class->methods.datas[idx].obj);
60         }
61         idx++;
62     }
63
64     //标灰类名
65     grayObject(vm, (ObjHeader*)class->name);
66
67     //累计类大小
68     vm->allocatedBytes += sizeof(Class);
69     vm->allocatedBytes += sizeof(Method) * class->methods.capacity;
70 }
71
72 //标灰闭包
73 static void blackClosure(VM* vm, ObjClosure* objClosure) {
74     //标灰闭包中的函数
75     grayObject(vm, (ObjHeader*)objClosure->fn);
76
77     //标灰包中的 upvalue
78     uint32_t idx = 0;
```




```
79      while (idx < objClosure->fn->upvalueNum) {
80          grayObject(vm, (ObjHeader*)objClosure->upvalues[idx]);
81          idx++;
82      }
83
84      //累计闭包大小
85      vm->allocatedBytes += sizeof(ObjClosure);
86                  vm->allocatedBytes += sizeof(ObjUpvalue*) * objClosure->fn->upvalueNum;
87  }
88
```

先介绍 GC 的一些底层函数，先有个大概印象即可，后面的标记阶段会大量用到这些函数，到时候就懂了。

函数 grayObject 用来标灰 obj，也就是把 obj 标为灰色，本质上就是收集到数组 vm->grays.grayObjects 中。在对象头 ObjHeader 中有个成员是 isDark，之前一直没用到它，在 GC 中是该成员派上用场的时候，如果对象头的 isDark 的值为 true，表示该对象为黑色，即保留对象。第 13 行判断若 obj 为空或 obj->isDark 为 true 则可直接返回。否则第 16 行将其 isDark 置为 true 表示该对象为黑色。说明一下，尽管函数是 grayObject，似乎是专门用来标灰的，但我们没有显式标出灰色和白色，原因已经在前面说过了，只要具备标黑的能力就行了。另外，灰色对象其实就是待保留的对象（黑色对象），因为标记—清扫回收算法就是从白色对象中挑选那些需要保留的对象，只是因为并未处理完，所以才将其置为中间态——灰色，以表示该对象处于处理过程中，最终标记过程完成后这些灰色对象就是黑色对象。所以 grayXXX 系列函数既用来处理灰色对象，也是后面标黑系列函数 blackXXX 的所封装的核心函数，这就是在 grayObject 函数中设置对象成员 isDark 的原因。第 19 行判断是否需要扩容，第 26 行将 obj 写到数组 grayObjects 中，并同时使 vm->grays.count 加 1 以更新下一个写入的位置。

函数 grayValue 用来标灰参数 value，因为 GC 用来回收对象，因此如果 value 不是对象就返回。本函数是调用 grayObject 完成标灰。

函数 grayBuffer 用来标灰参数 buffer 中 datas 数组中的所有 value。用 while 循环遍历 datas 数组，用 grayValue 处理每个元素。

函数 blackClass 用来标黑参数 class，调用 grayObject 函数分别标灰 class 的 meta 类、基类、所有方法及类名。最后是累计类的大小，目的是避免影响下次触发 GC 时的准确性。原因是在解释器运行中，类本身也要占用内存，它们是运行的必要数据结构，所以不能释放，因此它们所占的内存必须要算在解释器占用的内存量上，故累计其内存大小到 vm->allocatedBytes。

函数 blackClosure 用来标灰闭包 objClosure。闭包中需要标灰的成员对象有 objClosure->fn 和数组 objClosure->upvalues 中的 upvalue。最后再累计闭包的大小，原因同函数 blackClass。

stepByStep/c9/a/gc/gc.c (2/5)

```
89 //标黑 objThread
90 static void blackThread(VM* vm, ObjThread* objThread) {
91      //标灰 frame
92      uint32_t idx = 0;
93      while (idx < objThread->usedFrameNum) {
94          grayObject(vm, (ObjHeader*)objThread->frames[idx].closure);
95          idx++;
96      }
97
```




```
98      //标灰运行时栈中每个 slot
99      Value* slot =  objThread->stack;
100     while (slot < objThread->esp) {
101        grayValue(vm, *slot);
102        slot++;
103     }
104
105     //标灰本线程中所有的 upvalue
106     ObjUpvalue* upvalue = objThread->openUpvalues;
107     while (upvalue != NULL) {
108        grayObject(vm, (ObjHeader*)upvalue);
109        upvalue = upvalue->next;
110     }
111
112     //标灰 caller
113     grayObject(vm, (ObjHeader*)objThread->caller);
114     grayValue(vm, objThread->errorObj);
115
116     //累计线程大小
117     vm->allocatedBytes += sizeof(ObjThread);
118     vm->allocatedBytes += objThread->frameCapacity * sizeof(Frame);
119     vm->allocatedBytes += objThread->stackCapacity * sizeof(Value);
120 }
121
122 //标黑 fn
123 static void blackFn(VM* vm, ObjFn* fn) {
124     //标灰常量
125     grayBuffer(vm, &fn->constants);
126
127     //累计 Objfn 的空间
128     vm->allocatedBytes += sizeof(ObjFn);
129     vm->allocatedBytes += sizeof(uint8_t) * fn->instrStream.capacity;
130     vm->allocatedBytes += sizeof(Value) * fn->constants.capacity;
131
132 #if DEBUG
133     //再加上 debug 信息占用的内存
134     vm->allocatedBytes += sizeof(Int) * fn->instrStream.capacity;
135 #endif
136 }
137
138 //标黑 objInstance
139 static void blackInstance(VM* vm, ObjInstance* objInstance) {
140     //标灰元类
141     grayObject(vm, (ObjHeader*)objInstance->objHeader.class);
142
143     //标灰实例中所有域,域的个数在 class->fieldNum
144     uint32_t idx = 0;
145     while (idx < objInstance->objHeader.class->fieldNum) {
146        grayValue(vm, objInstance->fields[idx]);
147        idx++;
```




```
148     }
149
150     //累计 objInstance 空间
151     vm->allocatedBytes += sizeof(ObjInstance);
152     vm->allocatedBytes += sizeof(Value) * objInstance->objHeader.class->fieldNum;
153 }
154
155 //标黑 objList
156 static void blackList(VM* vm, ObjList* objList) {
157     //标灰 list 的 elements
158     grayBuffer(vm, &objList->elements);
159
160     //累计 objList 大小
161     vm->allocatedBytes += sizeof(ObjList);
162     vm->allocatedBytes += sizeof(Value) * objList->elements.capacity;
163 }
164
```

函数 blackThread 用于标黑形参 objThread。位于线程中的对象也要标灰，包括 objThread->frames 中所有 frame，运行时栈中每个 slot 中的对象，线程中已打开的 upvalue，即链表 objThread->openUpvalues 中所有节点，还要标灰线程的主调方即 objThread->caller，以及 errorObj 对象。最后累计线程大小。

函数 blackFn 用于标黑函数对象 fn。主要是标黑常量 fn->constants，然后累计函数对象大小。如果正处于 Debug 阶段，那么要加上所有指令的源码行号的尺寸。还记得吗，FnDebug 是内嵌在 ObjFn 中的结构，专门用于调试，里面记录了函数名及指令所对应脚本源码的行号，行号数组 lineNo 是 Int（int）型。在调式状态下我们在 writeByte 中把每条指令的行号都追加进了数组 lineNo 中，因此要用行号大小即 sizeof(Int)乘以函数 fn 中的指令数以获得所有指令行号的总大小。实际指令数是 fn->instrStream.count，这里用 fn->instrStream.capacity 做保守估算。

函数 blackInstance 用于标黑实例对象 objInstance。实例对象中要先标黑对象的元类，然后标黑所有域（成员属性）。最后累计对象内存大小。

函数 blackList 用于标黑列表对象 objList。先调用 grayBuffer 标灰列表对象中所有元素，最后累计列表对象的大小。

stepByStep/c9/a/gc/gc.c (3/5)

```
165 //标黑 objMap
166 static void blackMap(VM* vm, ObjMap* objMap) {
167     //标灰所有 entry
168     uint32_t idx = 0;
169     while (idx < objMap->capacity) {
170         Entry* entry = &objMap->entries[idx];
171         //跳过无效的 entry
172         if (!VALUE_IS_UNDEFINED(entry->key)) {
173             grayValue(vm, entry->key);
174             grayValue(vm, entry->value);
175         }
176         idx++;
177     }
```




```
178
179     //累计 ObjMap 大小
180     vm->allocatedBytes += sizeof(ObjMap);
181     vm->allocatedBytes += sizeof(Entry) * objMap->capacity;
182 }
183
184 //标黑 objModule
185 static void blackModule(VM* vm, ObjModule* objModule) {
186     //标灰模块中所有模块变量
187     uint32_t idx = 0;
188     while (idx < objModule->moduleVarValue.count) {
189        grayValue(vm, objModule->moduleVarValue.datas[idx]);
190        idx++;
191     }
192
193     //标灰模块名
194     grayObject(vm, (ObjHeader*)objModule->name);
195
196     //累计 ObjModule 大小
197     vm->allocatedBytes += sizeof(ObjModule);
198     vm->allocatedBytes+=sizeof(String)*objModule->moduleVarName.capacity;
199     vm->allocatedBytes+=sizeof(Value)*objModule->moduleVarValue.capacity;
200 }
201
202 //标黑 range
203 static void blackRange(VM* vm) {
204     //ObjRange 中没有大数据,只有 from 和 to,
205     //其空间属于 sizeof(ObjRange),因此不用额外标记
206     vm->allocatedBytes += sizeof(ObjRange);
207 }
208
209 //标黑 objString
210 static void blackString(VM* vm, ObjString* objString) {
211     //累计 ObjString 空间 +1 是结尾的'\0'
212     vm->allocatedBytes += sizeof(ObjString) + objString->value.length + 1;
213 }
214
215 //标黑 objUpvalue
216 static void blackUpvalue(VM* vm, ObjUpvalue* objUpvalue) {
217     //标灰 objUpvalue 的 closedUpvalue
218     grayValue(vm, objUpvalue->closedUpvalue);
219
220     //累计 objUpvalue 大小
221     vm->allocatedBytes += sizeof(ObjUpvalue);
222 }
223
224 //标黑 obj
225 static void blackObject(VM* vm, ObjHeader* obj) {
226 #ifdef DEBUG
227    printf("mark ");
```




```
228     dumpValue(OBJ_TO_VALUE(obj));
229     printf(" @ %p\n", obj);
230 #endif
231 //根据对象类型分别标黑
232     switch (obj->type) {
233        case OT_CLASS:
234           blackClass(vm, (Class*)obj);
235           break;
236
237        case OT_CLOSURE:
238           blackClosure(vm, (ObjClosure*)obj);
239           break;
240
241        case OT_THREAD:
242           blackThread(vm, (ObjThread*)obj);
243           break;
244
245        case OT_FUNCTION:
246           blackFn(vm, (ObjFn*)obj);
247           break;
248
249        case OT_INSTANCE:
250           blackInstance(vm, (ObjInstance*)obj);
251           break;
252
253        case OT_LIST:
254           blackList(vm, (ObjList*)obj);
255           break;
256
257        case OT_MAP:
258           blackMap(vm, (ObjMap*)obj);
259           break;
260
261        case OT_MODULE:
262           blackModule(vm, (ObjModule*)obj);
263           break;
264
265        case OT_RANGE:
266           blackRange(vm);
267           break;
268
269        case OT_STRING:
270           blackString(vm, (ObjString*)obj);
271           break;
272
273        case OT_UPVALUE:
274           blackUpvalue(vm, (ObjUpvalue*)obj);
275           break;
276     }
```




```
277 }
278
```

函数 blackMap 用于标黑 objMap，这包括 map 中所有 entry，即 key-value 对儿，因此循环调用 grayValue 处理 key 和 value，最后再累计 ObjMap 对象大小。

函数 blackModule 用于标黑 objModule。先标灰模块中的所有模块变量，再标灰模块名，最后累计模块对象 objModule 大小。

函数 blackRange 用于标黑 ObjRange 对象，本函数并没有形参，因为 ObjRange 对象中没有对象，只有表示一段范围的 from 和 to，因此只累计 ObjRange 大小即可。

函数 blackString 用于标黑形参 objString，累计其中的字符串大小。

函数 blackUpvalue 用于标黑形参 objUpvalue，先标黑其中的 closedUpvalue，再累计 objUpvalue 大小。

函数 blackObject 用于标黑形参 obj，函数开头的是调试状态下的输出，暂时先不管，后面实现 debug 工具时再看。接下来就是根据对象类型分别标黑。

stepByStep/c9/a/gc/gc.c (4/5)

```
279 //标黑那些已经标灰的对象，即保留那些标灰的对象
280 static void blackObjectInGray(VM* vm) {
281 //所有要保留的对象都已经收集到了 vm->grays.grayObjects 中，
282 //现在逐一标黑
283     while (vm->grays.count > 0) {
284         ObjHeader* objHeader = vm->grays.grayObjects[--vm->grays.count];
285         blackObject(vm, objHeader);
286     }
287 }
288
289 //释放 obj 自身及其占用的内存
290 void freeObject(VM* vm, ObjHeader* obj) {
291 #ifdef DEBUG
292     printf("free ");
293     dumpValue(OBJ_TO_VALUE(obj));
294     printf(" @ %p\n", obj);
295 #endif
296
297 //根据对象类型分别处理
298     switch (obj->type) {
299         case OT_CLASS:
300             MethodBufferClear(vm, &((Class*)obj)->methods);
301             break;
302
303         case OT_THREAD: {
304             ObjThread* objThread = (ObjThread*)obj;
305             DEALLOCATE(vm, objThread->frames);
306             DEALLOCATE(vm, objThread->stack);
307             break;
308         }
309
310         case OT_FUNCTION: {
311             ObjFn* fn = (ObjFn*)obj;
```




```
312             ValueBufferClear(vm, &fn->constants);
313             ByteBufferClear(vm, &fn->instrStream);
314         #if DEBUG
315             IntBufferClear(vm, &fn->debug->lineNo);
316             DEALLOCATE(vm, fn->debug->fnName);
317             DEALLOCATE(vm, fn->debug);
318         #endif
319             break;
320         }
321
322         case OT_LIST:
323             ValueBufferClear(vm, &((ObjList*)obj)->elements);
324             break;
325
326         case OT_MAP:
327             DEALLOCATE(vm, ((ObjMap*)obj)->entries);
328             break;
329
330         case OT_MODULE:
331             StringBufferClear(vm, &((ObjModule*)obj)->moduleVarName);
332             ValueBufferClear(vm, &((ObjModule*)obj)->moduleVarValue);
333             break;
334
335          case OT_STRING:
336          case OT_RANGE:
337          case OT_CLOSURE:
338          case OT_INSTANCE:
339          case OT_UPVALUE:
340            break;
341     }
342
343     //最后再释放自己
344     DEALLOCATE(vm, obj);
345 }
346
```

函数 blackObjectInGray 用于标黑那些已经标灰的对象，也就是保留那些标灰的对象，使其成为黑色对象。在调用本函数之前，所有灰色对象已经被收集到了 vm->grays.grayObjects 中，因此刚进入本函数时，vm->grays.grayObjects 中的对象全是根对象，后面介绍函数 startGC 时会细说。本函数其实就是标黑数组 vm->grays.grayObjects 中的每个对象，在执行过程中会间接调用到 grayObject 函数（因为函数 grayObject 是很多 blackXXX 系函数所封装的核心函数，而 blackObject 又封装的是这些 blackXXX 系函数），大家可以往回翻看函数 grayObject 的实现，该函数的结尾是把标灰的对象 obj 追加到数组 vm->grays.grayObjects 中，即代码 vm->grays.grayObjects[vm->grays.count++] = obj;，总之由于 blackObject 间接调用了 grayObject，从而导致不断往数组 vm->grays.grayObjects 中追加新的待保留的对象，因此在循环的开始阶段此数组的长度 vm->grays.count 的值会逐渐增大，然后随着循环的进行，第 284 行不断使 vm->grays.count 的值自减 1，所以在整个循环的过程中，数组 vm->grays.grayObjects 的长度 vm->grays.count 会呈现一种波浪线的发展趋势，即一会长上去，一会儿又降下来。说这些是想告诉大家，尽管最初数组 vm->grays.grayObjects



中的待保留对象（全是根对象）很少，但在此 while 循环开始后，很多待保留的对象（如那些未过生存期的对象）都会源源不断地通过 grayObject 函数被加到数组 vm->grays.grayObjects 中，这是用循环实现的递归（我们之前说过，“标记—清扫回收算法是在对象可达性‘递归’定义下最简单的垃圾回收方案”，当时强调了“递归”二字，本处正是此思想的实现），然后在循环中数组 vm->grays.grayObjects 中的所有对象都通过函数 blackObject 被标黑，从而得到了保留。因此数组 vm->grays.grayObjects 中的对象只是在 while 循环开始前是根对象，循环开始后就不再是纯粹的根对象了，全是等着被标黑的、待保留的对象。执行完后，vm->grays.grayObjects 中所有对象均变黑色，因为 vm->grays.grayObjects 中记录的只是对象地址，和“所有已分配对象链表” vm->allObjects 中指向的是相同的对象实体，因此执行完后，vm->allObjects 中所有对象要么是白色（可释放），要么是黑色（要保留），总之没有中间状态即灰色对象。vm->grays.grayObjects 是根对象集合，下面的 while 循环以此根集合为起始，进行深度优先标黑，为什么是深度优先呢？因为标黑函数 blackObject 中也是调用 grayObject，而 grayObject 会往 vm->grays.grayObjects 中加入新的保留对象。这里的 while 循环条件是 vm->grays.count > 0，循环体中是用--vm->grays.count 去索引数组 vm->grays.grayObjects，也就是从后往前处理，那么在每次循环中会先处理最新加入到数组 vm->grays.grayObjects 尾部的对象，因此实现了深度优先的标黑过程。

函数 freeObject 用来释放 obj 自身及其占用的内存，这是回收器释放垃圾对象时调用的函数。函数开头也是调试信息，后面就是根据不同对象类型分别释放相应的内存资源，大家自己看下吧。

stepByStep/c9/a/gc/gc.c (5/5)

```
347 //立即运行垃圾回收器去释放未用的内存
348 void startGC(VM* vm) {
349 #ifdef DEBUG
350     double startTime = (double)clock() / CLOCKS_PER_SEC;
351     uint32_t before = vm->allocatedBytes;
352     printf("-- gc  before:%d   nextGC:%d  vm:%p  --\n",
353            before, vm->config.nextGC, vm);
354 #endif
355 //  标记阶段:标记需要保留的对象
356
357     //将 allocatedBytes 置 0，便于精确统计回收后的总分配内存大小
358     vm->allocatedBytes = 0;
359
360     //allModules 不能被释放
361     grayObject(vm, (ObjHeader*)vm->allModules);
362
363     //标灰 tmpRoots 数组中的对象(不可达但是不想被回收,白名单)
364     uint32_t idx = 0;
365     while (idx < vm->tmpRootNum) {
366        grayObject(vm, vm->tmpRoots[idx]);
367        idx++;
368     }
369
370     //标灰当前线程,不能被回收
371     grayObject(vm, (ObjHeader*)vm->curThread);
372
373     //编译过程中若申请的内存过高就标灰编译单元
374     if (vm->curParser != NULL) {
```

```
375         ASSERT(vm->curParser->curCompileUnit != NULL,
376                "grayCompileUnit only be called while compiling!");
377         grayCompileUnit(vm, vm->curParser->curCompileUnit);
378     }
379
380     //置黑所有灰色对象(保留的对象)
381     blackObjectInGray(vm);
382
383 //  清扫阶段:回收白色对象(垃圾对象)
384
385     ObjHeader** obj = &vm->allObjects;
386     while (*obj != NULL) {
387         //回收白色对象
388         if (!((*obj)->isDark)) {
389             ObjHeader* unreached = *obj;
390             *obj = unreached->next;
391             freeObject(vm, unreached);
392         } else {
393             //如果已经是黑色对象,为了下一次 gc 重新判定,
394             //现在将其恢复为未标记状态,避免永远不被回收
395             (*obj)->isDark = false;
396             obj = &(*obj)->next;
397         }
398     }
399
400     //更新下一次触发 gc 的阈值
401     vm->config.nextGC = vm->allocatedBytes * vm->config.heapGrowthFactor;
402     if (vm->config.nextGC < vm->config.minHeapSize) {
403         vm->config.nextGC = vm->config.minHeapSize;
404     }
405
406 #ifdef DEBUG
407     double elapsed = ((double)clock() / CLOCKS_PER_SEC) - startTime;
408     printf("GC %lu before, %lu after (%lu collected), next at %lu. take %.3fs.\n",
409            (unsigned long)before,
410            (unsigned long)vm->allocatedBytes,
411            (unsigned long)(before - vm->allocatedBytes),
412            (unsigned long)vm->config.nextGC,
413            elapsed);
414 #endif
415 }
```

函数 startGC 用于运行垃圾回收器去释放未用的内存。函数开头是一些调试信息。startGC 是标记—清扫回收算法的入口，也是其具体的实现，因此 startGC 中分为标记和清扫两阶段。第 355 行先是进行标记阶段，标记那些需要保留的对象，前面说过了，肯定是先标记根对象。在进行之前先将 vm->allocatedBytes 清零，目的是为了精确统计垃圾回收后解释器所占用的内存，由前面 blackXXX 系列的函数中累计内存大小的部分代码来重新统计 vm->allocatedBytes 大小。先是标灰 vm->allModules，然后再标灰白名单数组 tmpRoots 中的临时根对象，随后标灰当前线程 vm->curThread。GC 是在任意时刻触发的，触发条件有两种（后面会知道），一种是系统分配的内存量



vm->allocatedBytes 到达阈值 vm->config.nextGC，另一种是由脚本代码中 System.gc()函数主动触发。编译过程中也要分配大量内存，因此有可能会导致内存分配量到达了触发 GC 的阈值。第 374 行，如果是在编译过程中触发 GC，还要调用函数 grayCompileUnit 标灰编译单元，避免编译单元 CompileUnit 被回收而编译失败。至此 vm->grays.grayObjects 中已经有了 vm->allModules、一些临时根对象和 vm->curThread，如果是编译中也有触发 GC，那么 vm->grays.grayObjects 中还包括 vm->curParser->curCompileUnit。在标黑（由函数 blackObjectInGray 完成）阶段开始前，vm->grays.grayObjects 中的是根对象集合，下面以此根集合为起始，调用函数 blackObjectInGray 进行深度优先标黑。“深度优先”是指根据对象类型，深入处理该对象内部的组成对象（比如线程对象中包括 upvalue 对象，upvalue 对象就是线程对象的组成对象），也就是说，上面只是把根对象即某个对象的最上层标记了，第 381 行调用 blackObjectInGray 函数以这些根对象为入口，进入其内层去标记其内部的组成对象。在 blackObjectInGray 内部会间接调用到 grayObject 函数，我们已经说过了，grayObject 会继续往 vm->grays.grayObjects 中追加灰色对象，此阶段的灰色对象就是黑色对象，因此我们说，vm->grays.grayObjects 中的成员只是在最初是根对象，这个“最初”就是调用 blackObjectInGray 之前，确切地说是 blackObjectInGray 函数中那个 while 循环之前。之后第 383 行便是清扫阶段了，即回收那些白色对象。对象的 isDark 成员若为 true 则表示是黑色对象。第 386 行开始循环遍历所有对象链表 vm->allObjects，判断对象的 isDark 是否为 true，若不为 true 表示是白色对象，即垃圾对象，则调用 freeObject 将其回收。否则已经是黑色对象了，为了下一次 GC 的重新判定，现在将其恢复为未标记状态 false 以表示默认是白色对象，这样可避免永远不被回收。接下来就是更新下一次触发 GC 的阈值，这里利用了虚拟机配置中的堆生长因子 heapGrowthFactor 来计算下一次触发 GC 的内存大小。最后是一些调试信息。

到这儿有关 GC 的实现已经完成了，后面还有一些 GC 相关调用的工作，下节补充。

9.6 添加临时根对象与触发 GC

GC 的触发条件有两种，一种是系统分配的内存量 vm->allocatedBytes 到达阈值 vm->config.nextGC，另一种是由脚本代码中 System.gc()函数主动触发（这两种的实现马上就会看到）。也就是说 GC 是在任何时间都可触发的，如果解释器在运行过程中创建了对象，并且不需要它们被 GC 回收，我们就要调用函数 pushTmpRoot 将其放到临时根对象集合即数组 tmpRoots 中，因为只是希望临时保护这些对象，之后就可以参与 GC 的回收淘汰过程，因此还需要调用与之匹配的 popTmpRoot 函数将这些对象从 tmpRoots 中删除。也就是说，我们需要在已经完成的那些代码文件中添加 pushTmpRoot 和 popTmpRoot 这两个函数的调用，注意，这对儿函数必须成对匹配地出现。现在我已经把这对儿函数添加到以下列出的源码文件的相关函数中了，因为这对儿函数并不影响所在函数的逻辑，因此具体代码就不单独列出了，也不好列，因为就是一两句，因此大家自己看下源文件吧。新增的 pushTmpRoot 和 popTmpRoot 这对儿函数调用的函数如表 9-1 所列。

表 9-1　　pushTmpRoot 和 popTmpRoot 调用的函数

文件	函数
compiler/compiler.c	addConstant defineModuleVar
object/class.c	newRawClass newClass
object/meta_obj.c	newObjModule

续表

文件	函数
object/obj_list.c	insertElement removeElement
object/obj_map.c	removeKey
vm/core.c	loadModule buildCore

接下来就是添加触发 GC 的代码。

stepByStep/c9/a/include/utils.c

```
 1 #include "utils.h"
 2 #include "vm.h"
 3 #include "parser.h"
 4 #include <stdlib.h>
 5 #include <stdarg.h>
 6 #include "gc.h"
 7
 8 //内存管理 3 种功能:
 9 //   1. 申请内存
10 //   2. 修改空间大小
11 //   3. 释放内存
12 void* memManager(VM* vm, void* ptr, uint32_t oldSize, uint32_t newSize) {
13     //累计系统分配的总内存
14     vm->allocatedBytes += newSize - oldSize;
15
16      //避免 realloc(NULL, 0)定义的新地址,此地址不能被释放
17     if (newSize == 0) {
18         free(ptr);
19         return NULL;
20     }
21
22     //在分配内存时若达到了 GC 触发的阈值则启动垃圾回收
23     if (newSize > 0 && vm->allocatedBytes > vm->config.nextGC) {
24         startGC(vm);
25     }
26
27     return realloc(ptr, newSize);
28 }
...略
```

函数 memManager 用分配内存，也就是说我们是在分配内存时判断当前已经申请的内存量是否已经大到触发 GC 的阈值。因此修改函数 memManager 代码，在第 23 行增加判断，如果当前内存的总分配量 vm->allocatedBytes 大于 GC 的触发阈值 vm->config.nextGC，就在第 24 行调用 startGC 进行垃圾回收。

还有一个原生函数要补充，就是 System.gc()，这是手动触发 GC 的方式，请看如下代码 core.c。

stepByStep/c9/a/vm/core.c

```
...略
1591 //System.clock: 返回以秒为单位的系统时钟
```

```
1592 static bool primSystemClock(VM* vm UNUSED, Value* args UNUSED) {
1593     RET_NUM((double)time(NULL));
1594 }
1595
1596 //System.gc(): 启动 gc
1597 static bool primSystemGC(VM* vm, Value* args) {
1598     startGC(vm);
1599     RET_NULL;
1600 }
…略
1741 //编译核心模块
1742 void buildCore(VM* vm) {
…略
1937 //system 类
1938 Class* systemClass = VALUE_TO_CLASS(getCoreClassValue(coreModule, "System"));
1939 PRIM_METHOD_BIND(systemClass->objHeader.class, "clock", primSystemClock);
1940 PRIM_METHOD_BIND(systemClass->objHeader.class, "gc()", primSystemGC);
1941 PRIM_METHOD_BIND(systemClass->objHeader.class, "importModule(_)",
    primSystemImportModule);
1942 PRIM_METHOD_BIND(systemClass->objHeader.class, "getModuleVariable(_,_)",
    primSystemGetModuleVariable);
1943 PRIM_METHOD_BIND(systemClass->objHeader.class, "writeString_(_)", primSyste-mWriteString);
…略
1954 }
```

第 1597 行新增原生函数 primSystemGC，其内部就是调用 startGC，然后返回 null。与 primSystemGC 关联的脚本方法是第 1940 行绑定的 System 类中的 gc()，即 System.gc()将会进行垃圾回收。

接下来是测试的阶段，我们已经添加了 GC 的代码，下面在 manager.sp 中添加 System.gc()调用，部分代码如下。

```
…略
 70  System.print(allEmployee[idx])
 71  idx = idx + 1
 72  }
 73
 74  System.gc()  //可以手动回收内存
 75
 76  var a = 3 + 5 > 9 - 3  ? "yes" : "no"
 77  if (a.endsWith("s")) {
…略
```

解释器执行，效果如下。

```
[work@work a]$ pwd
/home/work/workspace/stepByStep/c9/a
[work@work a]$ ./spr manager.sp
6000
8000
2
15000
```

```
16100
17300
number of employee:4
xm -> op
lz -> pm
lw -> manager
xh -> rd
averageHeight: 170.75
xh
xm
lw
lz
all employee are:[xh,xm,lw,lz,xl]
xh
xm
lw
lz
xl
1501456153
.dlrow ,olleh
[work@work a]$
```

当然只是增加了一句 System.gc()，没有任何输出，因此结果上看不出效果。由于脚本文件不大，其中创建的类并不多，因此在回收之后，解释器占用的内存量看不出明显差异，将来打开编译选项就看得明显了。

第 10 章　命令行及调试

10.1 释放虚拟机（本节源码 stepByStep/c10/a）

创建虚拟机是 newVM 函数完成的，在虚拟机完成使命之后，我们还要将其释放，释放虚拟机的函数是 freeVM，freeVM 也是在 vm.c 中实现，代码如下。

stepByStep/c10/a/vm/vm.c

```
...略
 58 //释放虚拟机 vm
 59 void freeVM(VM* vm) {
 60     ASSERT(vm->allMethodNames.count > 0, "VM have already been freed!");
 61
 62     //释放所有的对象
 63     ObjHeader* objHeader = vm->allObjects;
 64     while (objHeader != NULL) {
 65         //释放之前先备份下一个节点地址
 66         ObjHeader* next = objHeader->next;
 67         freeObject(vm, objHeader);
 68         objHeader = next;
 69     }
 70
 71     vm->grays.grayObjects = DEALLOCATE(vm, vm->grays.grayObjects);
 72     StringBufferClear(vm, &vm->allMethodNames);
 73     DEALLOCATE(vm, vm);
 74 }
...略
```

freeVM 的原理很简单，通过 while 循环遍历 vm->allObjects，调用 freeObject 逐一释放，然后调用 DEALLOCATE 释放 vm->grays.grayObjects，随后清空 vm->allMethodNames，最后调用 DEALLOCATE 释放虚拟机自身。

10.2 简单的命令行界面（本节源码 stepByStep/c10/a）

一直以来我们都是编译运行整个脚本文件，从未像 Python 那样在命令行中逐行运行代码，本节我们实现一个简单的命令行，这次修改的文件是如下的 cli.h 和 cli.c。

stepByStep/c10/a/cli/cli.h

```
  1 #ifndef _CLI_CLI_H
```

```
#define _CLI_CLI_H

#define VERSION 0.1.0
#define MAX_LINE_LEN 1024

#endif
```

宏 MAX_LINE_LEN 表示支持的单行最大长度为 1024，我们用它来存储用户键入的单行代码。宏 VERSION 表示命令行的版本为 0.1.0。

stepByStep/c10/a/cli/cli.c

```
…略
//运行命令行
static void runCli(void) {
    VM* vm = newVM();
    char sourceLine[MAX_LINE_LEN];
    printf("maque Version: 0.1\n");
    while (true) {
        printf(">>> ");

        //若读取失败或者键入 quit 就退出循环
        if (!fgets(sourceLine, MAX_LINE_LEN, stdin) ||
                memcmp(sourceLine, "quit", 4) == 0) {
            break;
        }
        executeModule(vm, OBJ_TO_VALUE(newObjString(vm, "cli", 3)), sourceLine);
    }
    freeVM(vm);
}

int main(int argc, const char** argv) {
    if (argc == 1) {
        runCli();
    } else {
        runFile(argv[1]);
    }
    return 0;
}
```

函数 runCli 就是我们的命令行的实现。实现比较简单，不支持多行编译，因此一次键入的代码必须符合完整语法，比如代码“for i ("abcd") {”会报错缺少右边的“}”。函数内部先调用 newVM 创建虚拟机实例，接下来的缓冲区 sourceLine 用来存储用户输入的代码，只存储一行。接下来的 while 是个死循环，先输出命令行提示符“>>> ”，然后读入用户的代码到 sourceLine，若键入的是“quit”就跳出循环，然后调用 freeVM 释放虚拟机，否则就调用 executeModule 编译执行 sourceLine 中的代码。

接下来在 main 函数中添加 runCli 的调用。下面是命令行运行的效果：

```
[work@work a]$ ./spr
maque Version: 0.1
>>> var a = ["a","b","c"]
>>> System.print(a)
```

```
[a,b,c]
>>> a.add("d")
>>> System.print(a[a.count-1])
d
>>> quit
[work@work a]$
```

10.3 调试（本节源码 stepByStep/c10/b）

本节是调试相关的代码，定义在 compiler 下 debug.h 和 debug.c 中，这部分我就不介绍了，大家需要的话自己看吧。说明一下，在使用 debug.c 中的代码时，记得在 makefile 中打开-DDEBUG 开关，也就是使用 makefile 中第 2 行的 CFLAGS 即可，把第 3 行的 CFLAGS 注释掉。

stepByStep/c10/b/compiler/debug.h

```
#ifdef DEBUG
    #ifndef _COMPILER_DEBUG_H
    #define _COMPILER_DEBUG_H
    #include "utils.h"
    #include "obj_fn.h"
    #include "obj_thread.h"

    void bindDebugFnName(VM* vm, FnDebug* fnDebug,
          const char* name, uint32_t length);
    void dumpValue(Value value);
    void dumpCode(VM* vm, ObjFn* fn);
    int dumpInstruction(VM* vm, ObjFn* fn, int i);
    void dumpStack(ObjThread* thread);
    void debugPrintStackTrace(VM* vm);
    #endif
#endif
```

stepByStep/c10/b/compiler/debug.c

```
#ifdef DEBUG
    #include <stdio.h>
    #include "debug.h"
    #include "vm.h"
    #include <string.h>

    //在 fnDebug 中绑定函数名
    void bindDebugFnName(VM* vm, FnDebug* fnDebug,
          const char* name, uint32_t length) {
       ASSERT(fnDebug->fnName == NULL, "debug.name has bound!");
       fnDebug->fnName = ALLOCATE_ARRAY(vm, char, length + 1);
       memcpy(fnDebug->fnName, name, length);
       fnDebug->fnName[length] = '\0';
    }

    //打印栈
    void dumpStack(ObjThread* thread) {
```

```
18      printf("(thread %p) stack:%p, esp:%p, slots:%ld ",
19            thread, thread->stack, thread->esp, thread->esp - thread->stack);
20      Value* slot = thread->stack;
21      while (slot < thread->esp) {
22         dumpValue(*slot);
23         printf(" | ");
24         slot++;
25      }
26      printf("\n");
27   }
28
29   //打印对象
30   static void dumpObject(ObjHeader* obj) {
31      switch (obj->type) {
32         case OT_CLASS:
33            printf("[class %s %p]", ((Class*)obj)->name->value.start, obj);
34            break;
35         case OT_CLOSURE:
36            printf("[closure %p]", obj);
37            break;
38         case OT_THREAD:
39            printf("[thread %p]", obj);
40            break;
41         case OT_FUNCTION:
42            printf("[fn %p]", obj);
43            break;
44         case OT_INSTANCE:
45            printf("[instance %p]", obj);
46            break;
47         case OT_LIST:
48            printf("[list %p]", obj);
49            break;
50         case OT_MAP:
51            printf("[map %p]", obj);
52            break;
53         case OT_MODULE:
54            printf("[module %p]", obj);
55            break;
56         case OT_RANGE:
57            printf("[range %p]", obj);
58            break;
59         case OT_STRING:
60            printf("%s", ((ObjString*)obj)->value.start);
61            break;
62         case OT_UPVALUE :
63            printf("[upvalue %p]", obj);
64            break;
65         default:
66            printf("[unknown object %d]", obj->type);
67            break;
```

```
    }
}

//打印 value
void dumpValue(Value value) {
    switch (value.type) {
        case VT_FALSE:     printf("false"); break;
        case VT_NULL:      printf("null"); break;
        case VT_NUM:       printf("%.14g", VALUE_TO_NUM(value)); break;
        case VT_TRUE:      printf("true"); break;
        case VT_OBJ:       dumpObject(VALUE_TO_OBJ(value)); break;
        case VT_UNDEFINED: NOT_REACHED();
    }
}

//打印一条指令
static int dumpOneInstruction(VM* vm, ObjFn* fn, int i, int* lastLine) {
    int start = i;
    uint8_t* bytecode = fn->instrStream.datas;
    OpCode opCode = (OpCode)bytecode[i];

    int lineNo = fn->debug->lineNo.datas[i];

    if (lastLine == NULL || *lastLine != lineNo) {
        printf("%4d:", lineNo);     //输出源码行号
        if (lastLine != NULL) {
            *lastLine = lineNo;
        }
    } else {
        //不用输出源码行了,还是输出的 lastLine 行的指令流,空出行号的位置即可
        printf("     ");
    }

    printf(" %04d  ", i++);    //输出指令流中的位置

    #define READ_BYTE() (bytecode[i++])
    #define READ_SHORT() (i += 2, (bytecode[i - 2] << 8) | bytecode[i - 1])

    #define BYTE_INSTRUCTION(name) \
        printf("%-16s %5d\n", name, READ_BYTE()); \
        break; \

    switch (opCode) {
        case OPCODE_LOAD_CONSTANT: {
            int constant = READ_SHORT();
            printf("%-16s %5d '", "LOAD_CONSTANT", constant);
            dumpValue(fn->constants.datas[constant]);
            printf("'\n");
            break;
        }
```

```
118
119         case OPCODE_PUSH_NULL:  printf("PUSH_NULL\n"); break;
120         case OPCODE_PUSH_FALSE: printf("PUSH_FALSE\n"); break;
121         case OPCODE_PUSH_TRUE:  printf("PUSH_TRUE\n"); break;
122
123         case OPCODE_LOAD_LOCAL_VAR: BYTE_INSTRUCTION("LOAD_LOCAL_VAR");
124         case OPCODE_STORE_LOCAL_VAR: BYTE_INSTRUCTION("STORE_LOCAL_VAR");
125         case OPCODE_LOAD_UPVALUE: BYTE_INSTRUCTION("LOAD_UPVALUE");
126         case OPCODE_STORE_UPVALUE: BYTE_INSTRUCTION("STORE_UPVALUE");
127
128         case OPCODE_LOAD_MODULE_VAR: {
129            int slot = READ_SHORT();
130            printf("%-16s %5d '%s'\n", "LOAD_MODULE_VAR", slot,
131                 fn->module->moduleVarName.datas[slot].str);
132            break;
133         }
134
135         case OPCODE_STORE_MODULE_VAR: {
136            int slot = READ_SHORT();
137            printf("%-16s %5d '%s'\n", "STORE_MODULE_VAR", slot,
138                 fn->module->moduleVarName.datas[slot].str);
139            break;
140         }
141
142         case OPCODE_LOAD_THIS_FIELD: BYTE_INSTRUCTION("LOAD_THIS_FIELD");
143         case OPCODE_STORE_THIS_FIELD: BYTE_INSTRUCTION("STORE_THIS_FIELD");
144         case OPCODE_LOAD_FIELD: BYTE_INSTRUCTION("LOAD_FIELD");
145         case OPCODE_STORE_FIELD: BYTE_INSTRUCTION("STORE_FIELD");
146
147         case OPCODE_POP: printf("POP\n"); break;
148
149         case OPCODE_CALL0:
150         case OPCODE_CALL1:
151         case OPCODE_CALL2:
152         case OPCODE_CALL3:
153         case OPCODE_CALL4:
154         case OPCODE_CALL5:
155         case OPCODE_CALL6:
156         case OPCODE_CALL7:
157         case OPCODE_CALL8:
158         case OPCODE_CALL9:
159         case OPCODE_CALL10:
160         case OPCODE_CALL11:
161         case OPCODE_CALL12:
162         case OPCODE_CALL13:
163         case OPCODE_CALL14:
164         case OPCODE_CALL15:
165         case OPCODE_CALL16: {
166            int numArgs = bytecode[i - 1] - OPCODE_CALL0;
167            int symbol = READ_SHORT();
```

```
168             printf("CALL%-11d %5d '%s'\n", numArgs, symbol,
169                     vm->allMethodNames.datas[symbol].str);
170             break;
171         }
172
173         case OPCODE_SUPER0:
174         case OPCODE_SUPER1:
175         case OPCODE_SUPER2:
176         case OPCODE_SUPER3:
177         case OPCODE_SUPER4:
178         case OPCODE_SUPER5:
179         case OPCODE_SUPER6:
180         case OPCODE_SUPER7:
181         case OPCODE_SUPER8:
182         case OPCODE_SUPER9:
183         case OPCODE_SUPER10:
184         case OPCODE_SUPER11:
185         case OPCODE_SUPER12:
186         case OPCODE_SUPER13:
187         case OPCODE_SUPER14:
188         case OPCODE_SUPER15:
189         case OPCODE_SUPER16: {
190             int numArgs = bytecode[i - 1] - OPCODE_SUPER0;
191             int symbol = READ_SHORT();
192             int superclass = READ_SHORT();
193             printf("SUPER%-10d %5d '%s' %5d\n", numArgs, symbol,
194                     vm->allMethodNames.datas[symbol].str, superclass);
195             break;
196         }
197
198         case OPCODE_JUMP: {
199             int offset = READ_SHORT();
200             printf("%-16s offset:%-5d abs:%d\n", "JUMP", offset, i + offset);
201             break;
202         }
203
204         case OPCODE_LOOP: {
205             int offset = READ_SHORT();
206             printf("%-16s offset:%-5d abs:%d\n", "LOOP", offset, i - offset);
207             break;
208         }
209
210         case OPCODE_JUMP_IF_FALSE: {
211             int offset = READ_SHORT();
212             printf("%-16s offset:%-5d abs:%d\n", "JUMP_IF_FALSE", offset, i + offset);
213             break;
214         }
215
216         case OPCODE_AND: {
217             int offset = READ_SHORT();
```

```
218             printf("%-16s offset:%-5d abs:%d\n", "AND", offset, i + offset);
219             break;
220         }
221
222         case OPCODE_OR: {
223             int offset = READ_SHORT();
224             printf("%-16s offset:%-5d abs:%d\n", "OR", offset, i + offset);
225             break;
226         }
227
228         case OPCODE_CLOSE_UPVALUE:
229             printf("CLOSE_UPVALUE\n");
230             break;
231
232         case OPCODE_RETURN:
233             printf("RETURN\n");
234             break;
235
236         case OPCODE_CREATE_CLOSURE: {
237             int constant = READ_SHORT();
238             printf("%-16s %5d ", "CREATE_CLOSURE", constant);
239             dumpValue(fn->constants.datas[constant]);
240             printf(" ");
241             ObjFn* loadedFn = VALUE_TO_OBJFN(fn->constants.datas[constant]);
242             uint32_t j;
243             for (j = 0; j < loadedFn->upvalueNum; j++) {
244                 int isLocal = READ_BYTE();
245                 int index = READ_BYTE();
246                 if (j > 0) printf(", ");
247                 printf("%s %d", isLocal ? "local" : "upvalue", index);
248             }
249             printf("\n");
250             break;
251         }
252
253         case OPCODE_CONSTRUCT:
254             printf("CONSTRUCT\n");
255             break;
256
257         case OPCODE_CREATE_CLASS: {
258             int numFields = READ_BYTE();
259             printf("%-16s %5d fields\n", "CREATE_CLASS", numFields);
260             break;
261         }
262
263         case OPCODE_INSTANCE_METHOD: {
264             int symbol = READ_SHORT();
265             printf("%-16s %5d '%s'\n", "INSTANCE_METHOD", symbol,
266                    vm->allMethodNames.datas[symbol].str);
267             break;
```

```
268         }
269
270         case OPCODE_STATIC_METHOD: {
271             int symbol = READ_SHORT();
272             printf("%-16s %5d '%s'\n", "STATIC_METHOD", symbol,
273                    vm->allMethodNames.datas[symbol].str);
274             break;
275         }
276
277         case OPCODE_END:
278             printf("END\n");
279             break;
280
281         default:
282             printf("UKNOWN! [%d]\n", bytecode[i - 1]);
283             break;
284     }
285
286     //返回指令占用的字节数
287     if (opCode == OPCODE_END) {
288         return -1;
289     }
290     return i - start;
291
292     #undef READ_BYTE
293     #undef READ_SHORT
294 }
295
296 //打印指令
297 void dumpInstructions(VM* vm, ObjFn* fn) {
298     printf("module:[%s]\tfunction:[%s]\n",
299            fn->module->name == NULL ? "<core>" : fn->module->name->value.start,
300            fn->debug->fnName);
301
302     int i = 0;
303     int lastLine = -1;
304     while (true) {
305         int offset = dumpOneInstruction(vm, fn, i, &lastLine);
306         if (offset == -1) break;
307         i += offset;
308     }
309     printf("\n");
310 }
311
312 #endif
```

下面在 makefile 中打开调试选项，看看调试环境下的输出吧，GC 的工作只有在调试下才能看到，不能辜负了这位默默无闻的兄弟。

如果用默认的 makefile 文件编译，那么上面所说的在 makefile 中打开调试选项，是指把第二行 CFLAGS 前的注释字符“#”去掉，然后把第三行 CFLAGS 前加上注释字符“#”，很简单对吧，

其实这两步修改就是在 CFLAGS 中增加了选项“–DDEBUG”，此选项表示定义一个名为 DEBUG 的宏，因为我们程序中有判断，如果定义了宏 DEBUG，就额外做一些工作，比如调用不同的函数或输出额外的信息等。注意，尽管这里定义的宏名是 DEBUG，但不要误以为宏名必须为 DEBUG，gcc 的选项-D 可以定义任意名字的宏，我们这里将宏名定义为 DEBUG，是因为我们在程序中把名为 DEBUG 的宏做为调试的开关，这两者是配合在一起的，如果-D 中定义的是 XXX，那么程序中就要判断是否定义了名为 XXX 的宏。如果你不想修改默认的 makefile，那需要单独生成一个 makefile。为此我单独准备了一份，目录中的文件 makefile.debug 就是专门用来调试的 makefile，说明一下，文件名“makefile.debug”是随意起的，不需要包含“makefile”或“debug”。由于此文件并不是 make 程序默认使用的 makefile，那么用 make 编译时就要通过-f 参数指定该文件，如下所示。

```
[work@work b]$ make -f makefile.debug
```

回车之后就是编译过程了，编译过程就不贴出了，下面是打开调试选项后，spr 执行脚本文件 manager.sp 的过程，如下所示。

```
[work@work b]$ ./spr manager.sp
-- gc  before:40   nextGC:0  vm:0x734010  --
GC 40 before, 0 after (40 collected), next at 0. take 0.000s.
6000
8000
2
15000
16100
17300
number of employee:4
xm -> op
lz -> pm
lw -> manager
xh -> rd
averageHeight: 170.75
xh
xm
lw
lz
all employee are:[xh,xm,lw,lz,xl]
xh
xm
lw
lz
xl
-- gc  before:121976   nextGC:10485760  vm:0x734010  --
mark [thread 0x750af0] @ 0x750af0
mark [thread 0x744c40] @ 0x744c40
mark [closure 0x744b80] @ 0x744b80
mark [fn 0x740620] @ 0x740620
mark male @ 0x742560
mark laozheng @ 0x742520
mark male @ 0x7424f0
```

```
mark laowang @ 0x74c690
mark [fn 0x74c330] @ 0x74c330
mark hello, world. @ 0x74caa0
mark error!!!!! @ 0x74ca60
mark s @ 0x74ca30
mark no @ 0x74ca00
mark yes @ 0x74c9d0
mark  @ 0x74c960
mark all employee are: @ 0x74c990
mark xl @ 0x74c930
mark lz @ 0x74c900
mark lw @ 0x74c8d0
mark xm @ 0x74c8a0
mark xh @ 0x74c4b0
mark  @ 0x74c480
mark averageHeight:  @ 0x74c440
mark lz @ 0x74c600
mark lw @ 0x74c5d0
mark xm @ 0x743cc0
mark xh @ 0x7426a0
mark  ->  @ 0x74c5a0
mark pm @ 0x74c520
mark lz @ 0x74c4f0
mark manager @ 0x74c660
mark lw @ 0x74c630
mark op @ 0x74c570
mark xm @ 0x745150
mark rd @ 0x74c3b0
mark xh @ 0x7441d0
mark number of employee: @ 0x74c400
mark [fn 0x745040] @ 0x745040
mark [class Employee 0x74f620] @ 0x74f620
mark Employee @ 0x7503b0
mark [closure 0x74f6f0] @ 0x74f6f0
mark [fn 0x74fa50] @ 0x74fa50
mark years old @ 0x74fc30
mark ,  @ 0x74fc70
mark , I am a  @ 0x74fba0
mark My name is  @ 0x74fb10
mark [closure 0x74d1e0] @ 0x74d1e0
mark [fn 0x74fe40] @ 0x74fe40
mark [closure 0x750480] @ 0x750480
mark [upvalue 0x7504b0] @ 0x7504b0
mark [fn 0x74d2d0] @ 0x74d2d0
mark [class object 0x734ab0] @ 0x734ab0
mark object @ 0x734b00
mark [class objectMeta 0x734c70] @ 0x734c70
mark objectMeta @ 0x734f50
mark [class class 0x734cc0] @ 0x734cc0
mark class @ 0x734c40
```

```
mark [class Employee metaclass 0x744df0] @ 0x744df0
mark Employee metaclass @ 0x74fa00
mark [closure 0x74d210] @ 0x74d210
mark [fn 0x74fec0] @ 0x74fec0
mark [closure 0x74d180] @ 0x74d180
mark [fn 0x74f8c0] @ 0x74f8c0
mark [fn 0x744f50] @ 0x744f50
mark [fn 0x7440f0] @ 0x7440f0
mark [fn 0x740590] @ 0x740590
mark Manager @ 0x7441a0
mark male @ 0x7444d0
mark xiaoming @ 0x743d10
mark female @ 0x7405f0
mark xiaohong @ 0x743d50
mark Employee @ 0x743c80
mark employee @ 0x744450
mark [class System 0x743fb0] @ 0x743fb0
mark System @ 0x744000
mark [class System metaclass 0x743e90] @ 0x743e90
mark System metaclass @ 0x743ee0
mark [closure 0x744500] @ 0x744500
mark [fn 0x73db40] @ 0x73db40
mark [closure 0x743f80] @ 0x743f80
mark [fn 0x73d8c0] @ 0x73d8c0
mark [closure 0x743f50] @ 0x743f50
mark [fn 0x73d7e0] @ 0x73d7e0
mark
 @ 0x73d950
mark [closure 0x744530] @ 0x744530
mark [fn 0x73dc60] @ 0x73dc60
mark [invalid toString] @ 0x73def0
mark [closure 0x743f20] @ 0x743f20
mark [fn 0x739550] @ 0x739550
mark
 @ 0x73d6f0
mark [closure 0x7440c0] @ 0x7440c0
mark [fn 0x7393b0] @ 0x7393b0
mark
 @ 0x7394b0
mark [list 0x756060] @ 0x756060
mark [map 0x755600] @ 0x755600
mark [map 0x750bd0] @ 0x750bd0
mark [closure 0x7505e0] @ 0x7505e0
mark [map 0x734110] @ 0x734110
mark [module 0x744cd0] @ 0x744cd0
mark employee @ 0x744d20
mark [class Range 0x743c30] @ 0x743c30
mark Range @ 0x743200
mark [closure 0x73f260] @ 0x73f260
mark [fn 0x737bf0] @ 0x737bf0
```

```
mark [closure 0x73f230] @ 0x73f230
mark [fn 0x737a90] @ 0x737a90
mark  @ 0x737b30
mark [closure 0x73f200] @ 0x73f200
mark [fn 0x736040] @ 0x736040
mark  @ 0x737300
mark [closure 0x73f1d0] @ 0x73f1d0
mark [fn 0x7371b0] @ 0x7371b0
mark Can't reduce an empty sequence. @ 0x737330
mark [closure 0x73ef40] @ 0x73ef40
mark [fn 0x736f70] @ 0x736f70
mark [closure 0x73ef10] @ 0x73ef10
mark [fn 0x736df0] @ 0x736df0
mark [closure 0x73eee0] @ 0x73eee0
mark [fn 0x7353b0] @ 0x7353b0
mark [closure 0x73eeb0] @ 0x73eeb0
mark [fn 0x7368d0] @ 0x7368d0
mark [closure 0x73ee80] @ 0x73ee80
mark [fn 0x7366a0] @ 0x7366a0
mark [closure 0x73ef90] @ 0x73ef90
mark [fn 0x736450] @ 0x736450
mark [closure 0x73ee20] @ 0x73ee20
mark [fn 0x735690] @ 0x735690
mark [closure 0x73edf0] @ 0x73edf0
mark [fn 0x735e70] @ 0x735e70
mark [closure 0x73edc0] @ 0x73edc0
mark [fn 0x735c80] @ 0x735c80
mark [closure 0x73ee50] @ 0x73ee50
mark [fn 0x735580] @ 0x735580
mark [class Range metaclass 0x743b10] @ 0x743b10
mark Range metaclass @ 0x743b60
mark [class MapValueSequence 0x743230] @ 0x743230
mark MapValueSequence @ 0x743280
mark [closure 0x7427f0] @ 0x7427f0
mark [fn 0x73cc70] @ 0x73cc70
mark [closure 0x7431d0] @ 0x7431d0
mark [fn 0x739220] @ 0x739220
mark [closure 0x7431a0] @ 0x7431a0
mark [fn 0x738b10] @ 0x738b10
mark [class MapValueSequence metaclass 0x743100] @ 0x743100
mark MapValueSequence metaclass @ 0x743150
mark [closure 0x7436d0] @ 0x7436d0
mark [fn 0x73cd80] @ 0x73cd80
mark [class MapKeySequence 0x742820] @ 0x742820
mark MapKeySequence @ 0x742870
mark [closure 0x742280] @ 0x742280
mark [fn 0x73c7f0] @ 0x73c7f0
mark [closure 0x7427c0] @ 0x7427c0
mark [fn 0x73cb00] @ 0x73cb00
mark [closure 0x742790] @ 0x742790
```




仅供非商业用途或交流学习使用


```
mark [fn 0x73c9d0] @ 0x73c9d0
mark [class MapKeySequence metaclass 0x7426f0] @ 0x7426f0
mark MapKeySequence metaclass @ 0x742740
mark [closure 0x742cc0] @ 0x742cc0
mark [fn 0x73c8e0] @ 0x73c8e0
mark [class Map 0x741d50] @ 0x741d50
mark Map @ 0x741da0
mark [closure 0x742220] @ 0x742220
mark [fn 0x738820] @ 0x738820
mark [closure 0x7422b0] @ 0x7422b0
mark [fn 0x73afd0] @ 0x73afd0
mark [closure 0x742250] @ 0x742250
mark [fn 0x738970] @ 0x738970
mark } @ 0x73c7c0
mark  @ 0x73c450
mark :  @ 0x73c420
mark  @ 0x73c4a0
mark ,  @ 0x73c360
mark { @ 0x738a50
mark [class Map metaclass 0x741c30] @ 0x741c30
mark Map metaclass @ 0x741c80
mark [class StringCodePointSequence 0x740840] @ 0x740840
mark StringCodePointSequence @ 0x740890
mark [closure 0x7407b0] @ 0x7407b0
mark [fn 0x73a290] @ 0x73a290
mark [closure 0x741380] @ 0x741380
mark [fn 0x73a0f0] @ 0x73a0f0
mark [closure 0x741170] @ 0x741170
mark [fn 0x73a5e0] @ 0x73a5e0
mark [closure 0x740810] @ 0x740810
mark [fn 0x73a4d0] @ 0x73a4d0
mark [closure 0x7407e0] @ 0x7407e0
mark [fn 0x73a3c0] @ 0x73a3c0
mark [class StringCodePointSequence metaclass 0x740710] @ 0x740710
mark StringCodePointSequence metaclass @ 0x740760
mark [closure 0x7408d0] @ 0x7408d0
mark [fn 0x73a1d0] @ 0x73a1d0
mark [class StringByteSequence 0x740680] @ 0x740680
mark StringByteSequence @ 0x7406d0
mark [closure 0x7401f0] @ 0x7401f0
mark [fn 0x739be0] @ 0x739be0
mark [closure 0x740920] @ 0x740920
mark [fn 0x7376c0] @ 0x7376c0
mark [closure 0x740190] @ 0x740190
mark [fn 0x739f70] @ 0x739f70
mark [closure 0x740160] @ 0x740160
mark [fn 0x739e60] @ 0x739e60
mark [closure 0x740130] @ 0x740130
mark [fn 0x739d30] @ 0x739d30
mark [class StringByteSequence metaclass 0x740090] @ 0x740090
```




```
mark StringByteSequence metaclass @ 0x7400e0
mark [closure 0x7401c0] @ 0x7401c0
mark [fn 0x739b20] @ 0x739b20
mark [class String 0x73ffe0] @ 0x73ffe0
mark String @ 0x73f9b0
mark [closure 0x740060] @ 0x740060
mark [fn 0x7375b0] @ 0x7375b0
mark  @ 0x737720
mark Count must be a non-negative integer. @ 0x739790
mark [closure 0x740030] @ 0x740030
mark [fn 0x7362f0] @ 0x7362f0
mark [closure 0x740240] @ 0x740240
mark [fn 0x738db0] @ 0x738db0
mark [class String metaclass 0x73fec0] @ 0x73fec0
mark String metaclass @ 0x73ff10
mark [class List 0x7412c0] @ 0x7412c0
mark List @ 0x741310
mark [closure 0x741de0] @ 0x741de0
mark [fn 0x73a700] @ 0x73a700
mark [closure 0x741c00] @ 0x741c00
mark [fn 0x73aea0] @ 0x73aea0
mark Count must be a non-negative integer. @ 0x73b0f0
mark [closure 0x741bd0] @ 0x741bd0
mark [fn 0x73ace0] @ 0x73ace0
mark [closure 0x741340] @ 0x741340
mark [fn 0x73a7f0] @ 0x73a7f0
mark ] @ 0x73aba0
mark , @ 0x73aa80
mark [ @ 0x73a850
mark [class List metaclass 0x7411a0] @ 0x7411a0
mark List metaclass @ 0x7411f0
mark [class WhereSequence 0x73f9e0] @ 0x73f9e0
mark WhereSequence @ 0x73fa30
mark [closure 0x73f380] @ 0x73f380
mark [fn 0x735940] @ 0x735940
mark [closure 0x73f980] @ 0x73f980
mark [fn 0x738ca0] @ 0x738ca0
mark [closure 0x73f950] @ 0x73f950
mark [fn 0x736cc0] @ 0x736cc0
mark [class WhereSequence metaclass 0x73f8c0] @ 0x73f8c0
mark WhereSequence metaclass @ 0x73f910
mark [closure 0x73fc80] @ 0x73fc80
mark [fn 0x736bd0] @ 0x736bd0
mark [class MapSequence 0x73f3b0] @ 0x73f3b0
mark MapSequence @ 0x73f400
mark [closure 0x73f650] @ 0x73f650
mark [fn 0x737f20] @ 0x737f20
mark [closure 0x73f350] @ 0x73f350
mark [fn 0x738230] @ 0x738230
mark [closure 0x73f320] @ 0x73f320
```

```
mark [fn 0x738120] @ 0x738120
mark [class MapSequence metaclass 0x73f290] @ 0x73f290
mark MapSequence metaclass @ 0x73f2e0
mark [closure 0x73f680] @ 0x73f680
mark [fn 0x738030] @ 0x738030
mark [class Sequence 0x73ed30] @ 0x73ed30
mark Sequence @ 0x73ed80
mark [class Sequence metaclass 0x73ec10] @ 0x73ec10
mark Sequence metaclass @ 0x73ec60
mark [class Thread 0x73eb00] @ 0x73eb00
mark Thread @ 0x73eb50
mark [class Thread metaclass 0x73e9e0] @ 0x73e9e0
mark Thread metaclass @ 0x73ea30
mark [class Fn 0x73e8d0] @ 0x73e8d0
mark Fn @ 0x73e920
mark [class Fn metaclass 0x73e7b0] @ 0x73e7b0
mark Fn metaclass @ 0x73e800
mark [class Num 0x73e6a0] @ 0x73e6a0
mark Num @ 0x73e6f0
mark [class Num metaclass 0x73e580] @ 0x73e580
mark Num metaclass @ 0x73e5d0
mark [class Bool 0x73e470] @ 0x73e470
mark Bool @ 0x73e4c0
mark [class Bool metaclass 0x73e350] @ 0x73e350
mark Bool metaclass @ 0x73e3a0
mark [class Null 0x73e270] @ 0x73e270
mark Null @ 0x73dcc0
mark [class Null metaclass 0x73e190] @ 0x73e190
mark Null metaclass @ 0x73de20
mark [module 0x740480] @ 0x740480
mark manager.sp @ 0x7404d0
mark [instance 0x7508d0] @ 0x7508d0
mark [instance 0x750750] @ 0x750750
mark [class Manager 0x750640] @ 0x750640
mark Manager @ 0x750690
mark [closure 0x7506f0] @ 0x7506f0
mark [closure 0x7506c0] @ 0x7506c0
mark [closure 0x7505b0] @ 0x7505b0
mark [class Manager metaclass 0x750520] @ 0x750520
mark Manager metaclass @ 0x750570
mark [closure 0x750720] @ 0x750720
mark [instance 0x750320] @ 0x750320
mark [instance 0x7503f0] @ 0x7503f0
mark manager.sp @ 0x7391c0
mark [module 0x734250] @ 0x734250
free all employee are:[xh,xm,lw,lz,xl] @ 0x756770
free all employee are:[xh,xm,lw,lz,xl] @ 0x756720
free all employee are:[xh,xm,lw,lz,xl] @ 0x7566d0
free all employee are: @ 0x756690
free all employee are: @ 0x756650
```

仅供非商业用途或交流学习使用


```
free [xh,xm,lw,lz,xl] @ 0x7565c0
free [xh,xm,lw,lz,xl @ 0x756580
free [xh,xm,lw,lz,xl @ 0x756540
free [ @ 0x756510
free [ @ 0x7564e0
free xh,xm,lw,lz,xl @ 0x756450
free xh,xm,lw,lz, @ 0x756410
free xh,xm,lw,lz @ 0x7563d0
free xh,xm,lw, @ 0x756390
free xh,xm,lw @ 0x750aa0
free xh,xm, @ 0x750a70
free xh,xm @ 0x750a40
free xh, @ 0x750a10
free xh @ 0x7509e0
free [list 0x756150] @ 0x756150
free [list 0x756120] @ 0x756120
free averageHeight: 170.75 @ 0x756020
free averageHeight: 170.75 @ 0x755fe0
free averageHeight: 170.75 @ 0x755fa0
free 170.75 @ 0x755f70
free averageHeight:  @ 0x755f30
free averageHeight:  @ 0x755ef0
free [list 0x755e70] @ 0x755e70
free [instance 0x755e40] @ 0x755e40
free xh -> rd @ 0x7555c0
free xh ->  @ 0x755590
free lw -> manager @ 0x755550
free lw ->  @ 0x7514c0
free lz -> pm @ 0x751480
free lz ->  @ 0x751450
free xm -> op @ 0x751410
free xm ->  @ 0x744c10
free [instance 0x744be0] @ 0x744be0
free number of employee:4 @ 0x750b90
free 4 @ 0x750b60
free 17300 @ 0x750940
free 16100 @ 0x744bb0
free 15000 @ 0x750610
free 2 @ 0x7504f0
free 8000 @ 0x750380
free 6000 @ 0x750450
free [thread 0x7502b0] @ 0x7502b0
free [closure 0x74f9d0] @ 0x74f9d0
free Employee @ 0x74d140
free [fn 0x74d760] @ 0x74d760
free [thread 0x73e120] @ 0x73e120
free [closure 0x73dbe0] @ 0x73dbe0
free System @ 0x739380
free Range @ 0x739350
free MapValueSequence @ 0x73cc30
```

仅供非商业用途或交流学习使用




```
free MapKeySequence @ 0x738ad0
free Map @ 0x73ab70
free List @ 0x73a660
free StringCodePointSequence @ 0x73a0b0
free StringByteSequence @ 0x7396e0
free String @ 0x737d80
free WhereSequence @ 0x738340
free MapSequence @ 0x737d40
free Sequence @ 0x735370
free Thread @ 0x735490
free Fn @ 0x7350b0
free Num @ 0x735550
free Bool @ 0x734c10
free Null @ 0x734e90
free [fn 0x734d30] @ 0x734d30
GC 121976 before, 93580 after (28396 collected), next at 1048576. take 0.000s.
1502006315
.dlrow ,olleh
free .dlrow ,olleh @ 0x737d40
free [range 0x734d90] @ 0x734d90
free 1502006315 @ 0x735370
free [list 0x756060] @ 0x756060
free [map 0x755600] @ 0x755600
free [map 0x750bd0] @ 0x750bd0
free [thread 0x750af0] @ 0x750af0
free [instance 0x7508d0] @ 0x7508d0
free [instance 0x750750] @ 0x750750
free [closure 0x7505e0] @ 0x7505e0
free [closure 0x7505b0] @ 0x7505b0
free [closure 0x750720] @ 0x750720
free [closure 0x7506f0] @ 0x7506f0
free [closure 0x7506c0] @ 0x7506c0
free Manager @ 0x750690
free [class Manager 0x750640] @ 0x750640
free Manager metaclass @ 0x750570
free [class Manager metaclass 0x750520] @ 0x750520
free [instance 0x750320] @ 0x750320
free [instance 0x7503f0] @ 0x7503f0
free [closure 0x74d210] @ 0x74d210
free [closure 0x74d1e0] @ 0x74d1e0
free [closure 0x74f6f0] @ 0x74f6f0
free [closure 0x74d180] @ 0x74d180
free [upvalue 0x7504b0] @ 0x7504b0
free [closure 0x750480] @ 0x750480
free Employee @ 0x7503b0
free [class Employee 0x74f620] @ 0x74f620
free Employee metaclass @ 0x74fa00
free [class Employee metaclass 0x744df0] @ 0x744df0
free [fn 0x74fec0] @ 0x74fec0
free [fn 0x74fe40] @ 0x74fe40
```




仅供非商业用途或交流学习使用




```
free years old @ 0x74fc30
free ,  @ 0x74fc70
free , I am a  @ 0x74fba0
free My name is  @ 0x74fb10
free [fn 0x74fa50] @ 0x74fa50
free [fn 0x74f8c0] @ 0x74f8c0
free [fn 0x74d2d0] @ 0x74d2d0
free employee @ 0x744d20
free [module 0x744cd0] @ 0x744cd0
free [thread 0x744c40] @ 0x744c40
free [closure 0x744b80] @ 0x744b80
free male @ 0x742560
free laozheng @ 0x742520
free male @ 0x7424f0
free laowang @ 0x74c690
free hello, world. @ 0x74caa0
free error!!!!! @ 0x74ca60
free s @ 0x74ca30
free no @ 0x74ca00
free yes @ 0x74c9d0
free  @ 0x74c960
free all employee are: @ 0x74c990
free xl @ 0x74c930
free lz @ 0x74c900
free lw @ 0x74c8d0
free xm @ 0x74c8a0
free xh @ 0x74c4b0
free  @ 0x74c480
free averageHeight:  @ 0x74c440
free lz @ 0x74c600
free lw @ 0x74c5d0
free xm @ 0x743cc0
free xh @ 0x7426a0
free  ->  @ 0x74c5a0
free pm @ 0x74c520
free lz @ 0x74c4f0
free manager @ 0x74c660
free lw @ 0x74c630
free op @ 0x74c570
free xm @ 0x745150
free rd @ 0x74c3b0
free xh @ 0x7441d0
free number of employee: @ 0x74c400
free [fn 0x74c330] @ 0x74c330
free [fn 0x745040] @ 0x745040
free [fn 0x744f50] @ 0x744f50
free [fn 0x7440f0] @ 0x7440f0
free [fn 0x740590] @ 0x740590
free Manager @ 0x7441a0
free male @ 0x7444d0
```



仅供非商业用途或交流学习使用




```
free xiaoming @ 0x743d10
free female @ 0x7405f0
free xiaohong @ 0x743d50
free Employee @ 0x743c80
free employee @ 0x744450
free [fn 0x740620] @ 0x740620
free manager.sp @ 0x7404d0
free [module 0x740480] @ 0x740480
free manager.sp @ 0x7391c0
free [closure 0x744530] @ 0x744530
free [closure 0x744500] @ 0x744500
free [closure 0x743f80] @ 0x743f80
free [closure 0x743f50] @ 0x743f50
free [closure 0x743f20] @ 0x743f20
free [closure 0x7440c0] @ 0x7440c0
free System @ 0x744000
free [class System 0x743fb0] @ 0x743fb0
free System metaclass @ 0x743ee0
free [class System metaclass 0x743e90] @ 0x743e90
free Range @ 0x743200
free [class Range 0x743c30] @ 0x743c30
free Range metaclass @ 0x743b60
free [class Range metaclass 0x743b10] @ 0x743b10
free [closure 0x7431d0] @ 0x7431d0
free [closure 0x7431a0] @ 0x7431a0
free [closure 0x7436d0] @ 0x7436d0
free [closure 0x7427f0] @ 0x7427f0
free MapValueSequence @ 0x743280
free [class MapValueSequence 0x743230] @ 0x743230
free MapValueSequence metaclass @ 0x743150
free [class MapValueSequence metaclass 0x743100] @ 0x743100
free [closure 0x7427c0] @ 0x7427c0
free [closure 0x742790] @ 0x742790
free [closure 0x742cc0] @ 0x742cc0
free [closure 0x742280] @ 0x742280
free MapKeySequence @ 0x742870
free [class MapKeySequence 0x742820] @ 0x742820
free MapKeySequence metaclass @ 0x742740
free [class MapKeySequence metaclass 0x7426f0] @ 0x7426f0
free [closure 0x742250] @ 0x742250
free [closure 0x742220] @ 0x742220
free [closure 0x7422b0] @ 0x7422b0
free Map @ 0x741da0
free [class Map 0x741d50] @ 0x741d50
free Map metaclass @ 0x741c80
free [class Map metaclass 0x741c30] @ 0x741c30
free [closure 0x741c00] @ 0x741c00
free [closure 0x741bd0] @ 0x741bd0
free [closure 0x741340] @ 0x741340
free [closure 0x741de0] @ 0x741de0
```



仅供非商业用途或交流学习使用




```
free List @ 0x741310
free [class List 0x7412c0] @ 0x7412c0
free List metaclass @ 0x7411f0
free [class List metaclass 0x7411a0] @ 0x7411a0
free [closure 0x741170] @ 0x741170
free [closure 0x740810] @ 0x740810
free [closure 0x7407e0] @ 0x7407e0
free [closure 0x7407b0] @ 0x7407b0
free [closure 0x7408d0] @ 0x7408d0
free [closure 0x741380] @ 0x741380
free StringCodePointSequence @ 0x740890
free [class StringCodePointSequence 0x740840] @ 0x740840
free StringCodePointSequence metaclass @ 0x740760
free [class StringCodePointSequence metaclass 0x740710] @ 0x740710
free [closure 0x740190] @ 0x740190
free [closure 0x740160] @ 0x740160
free [closure 0x740130] @ 0x740130
free [closure 0x7401f0] @ 0x7401f0
free [closure 0x7401c0] @ 0x7401c0
free [closure 0x740920] @ 0x740920
free StringByteSequence @ 0x7406d0
free [class StringByteSequence 0x740680] @ 0x740680
free StringByteSequence metaclass @ 0x7400e0
free [class StringByteSequence metaclass 0x740090] @ 0x740090
free [closure 0x740060] @ 0x740060
free [closure 0x740030] @ 0x740030
free [closure 0x740240] @ 0x740240
free String @ 0x73f9b0
free [class String 0x73ffe0] @ 0x73ffe0
free String metaclass @ 0x73ff10
free [class String metaclass 0x73fec0] @ 0x73fec0
free [closure 0x73f980] @ 0x73f980
free [closure 0x73f950] @ 0x73f950
free [closure 0x73fc80] @ 0x73fc80
free [closure 0x73f380] @ 0x73f380
free WhereSequence @ 0x73fa30
free [class WhereSequence 0x73f9e0] @ 0x73f9e0
free WhereSequence metaclass @ 0x73f910
free [class WhereSequence metaclass 0x73f8c0] @ 0x73f8c0
free [closure 0x73f350] @ 0x73f350
free [closure 0x73f320] @ 0x73f320
free [closure 0x73f680] @ 0x73f680
free [closure 0x73f650] @ 0x73f650
free MapSequence @ 0x73f400
free [class MapSequence 0x73f3b0] @ 0x73f3b0
free MapSequence metaclass @ 0x73f2e0
free [class MapSequence metaclass 0x73f290] @ 0x73f290
free [closure 0x73f260] @ 0x73f260
free [closure 0x73f230] @ 0x73f230
free [closure 0x73f200] @ 0x73f200
```



仅供非商业用途或交流学习使用




```
free [closure 0x73f1d0] @ 0x73f1d0
free [closure 0x73ef40] @ 0x73ef40
free [closure 0x73ef10] @ 0x73ef10
free [closure 0x73eee0] @ 0x73eee0
free [closure 0x73eeb0] @ 0x73eeb0
free [closure 0x73ee80] @ 0x73ee80
free [closure 0x73ef90] @ 0x73ef90
free [closure 0x73ee20] @ 0x73ee20
free [closure 0x73edf0] @ 0x73edf0
free [closure 0x73edc0] @ 0x73edc0
free [closure 0x73ee50] @ 0x73ee50
free Sequence @ 0x73ed80
free [class Sequence 0x73ed30] @ 0x73ed30
free Sequence metaclass @ 0x73ec60
free [class Sequence metaclass 0x73ec10] @ 0x73ec10
free Thread @ 0x73eb50
free [class Thread 0x73eb00] @ 0x73eb00
free Thread metaclass @ 0x73ea30
free [class Thread metaclass 0x73e9e0] @ 0x73e9e0
free Fn @ 0x73e920
free [class Fn 0x73e8d0] @ 0x73e8d0
free Fn metaclass @ 0x73e800
free [class Fn metaclass 0x73e7b0] @ 0x73e7b0
free Num @ 0x73e6f0
free [class Num 0x73e6a0] @ 0x73e6a0
free Num metaclass @ 0x73e5d0
free [class Num metaclass 0x73e580] @ 0x73e580
free Bool @ 0x73e4c0
free [class Bool 0x73e470] @ 0x73e470
free Bool metaclass @ 0x73e3a0
free [class Bool metaclass 0x73e350] @ 0x73e350
free Null @ 0x73dcc0
free [class Null 0x73e270] @ 0x73e270
free Null metaclass @ 0x73de20
free [class Null metaclass 0x73e190] @ 0x73e190
free [invalid toString] @ 0x73def0
free [fn 0x73dc60] @ 0x73dc60
free [fn 0x73db40] @ 0x73db40
free [fn 0x73d8c0] @ 0x73d8c0
free
 @ 0x73d950
free [fn 0x73d7e0] @ 0x73d7e0
free
 @ 0x73d6f0
free [fn 0x739550] @ 0x739550
free
 @ 0x7394b0
free [fn 0x7393b0] @ 0x7393b0
free [fn 0x739220] @ 0x739220
free [fn 0x738b10] @ 0x738b10
```




仅供非商业用途或交流学习使用




```
free [fn 0x73cd80] @ 0x73cd80
free [fn 0x73cc70] @ 0x73cc70
free [fn 0x73cb00] @ 0x73cb00
free [fn 0x73c9d0] @ 0x73c9d0
free [fn 0x73c8e0] @ 0x73c8e0
free [fn 0x73c7f0] @ 0x73c7f0
free } @ 0x73c7c0
free  @ 0x73c450
free :  @ 0x73c420
free  @ 0x73c4a0
free ,  @ 0x73c360
free { @ 0x738a50
free [fn 0x738970] @ 0x738970
free [fn 0x738820] @ 0x738820
free [fn 0x73afd0] @ 0x73afd0
free Count must be a non-negative integer. @ 0x73b0f0
free [fn 0x73aea0] @ 0x73aea0
free [fn 0x73ace0] @ 0x73ace0
free ] @ 0x73aba0
free , @ 0x73aa80
free [ @ 0x73a850
free [fn 0x73a7f0] @ 0x73a7f0
free [fn 0x73a700] @ 0x73a700
free [fn 0x73a5e0] @ 0x73a5e0
free [fn 0x73a4d0] @ 0x73a4d0
free [fn 0x73a3c0] @ 0x73a3c0
free [fn 0x73a290] @ 0x73a290
free [fn 0x73a1d0] @ 0x73a1d0
free [fn 0x73a0f0] @ 0x73a0f0
free [fn 0x739f70] @ 0x739f70
free [fn 0x739e60] @ 0x739e60
free [fn 0x739d30] @ 0x739d30
free [fn 0x739be0] @ 0x739be0
free [fn 0x739b20] @ 0x739b20
free [fn 0x7376c0] @ 0x7376c0
free  @ 0x737720
free Count must be a non-negative integer. @ 0x739790
free [fn 0x7375b0] @ 0x7375b0
free [fn 0x7362f0] @ 0x7362f0
free [fn 0x738db0] @ 0x738db0
free [fn 0x738ca0] @ 0x738ca0
free [fn 0x736cc0] @ 0x736cc0
free [fn 0x736bd0] @ 0x736bd0
free [fn 0x735940] @ 0x735940
free [fn 0x738230] @ 0x738230
free [fn 0x738120] @ 0x738120
free [fn 0x738030] @ 0x738030
free [fn 0x737f20] @ 0x737f20
free [fn 0x737bf0] @ 0x737bf0
free  @ 0x737b30
```



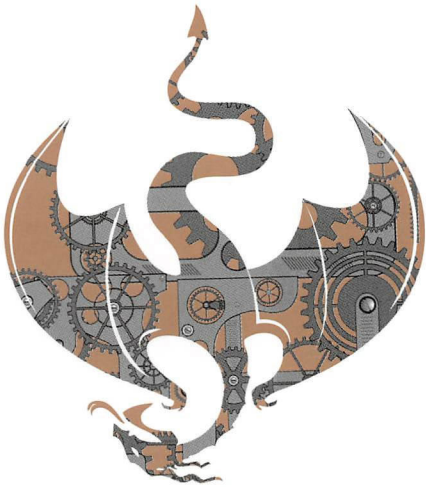

仅供非商业用途或交流学习使用




```
free [fn 0x737a90] @ 0x737a90
free  @ 0x737300
free [fn 0x736040] @ 0x736040
free Can't reduce an empty sequence. @ 0x737330
free [fn 0x7371b0] @ 0x7371b0
free [fn 0x736f70] @ 0x736f70
free [fn 0x736df0] @ 0x736df0
free [fn 0x7353b0] @ 0x7353b0
free [fn 0x7368d0] @ 0x7368d0
free [fn 0x7366a0] @ 0x7366a0
free [fn 0x736450] @ 0x736450
free [fn 0x735690] @ 0x735690
free [fn 0x735e70] @ 0x735e70
free [fn 0x735c80] @ 0x735c80
free [fn 0x735580] @ 0x735580
free objectMeta @ 0x734f50
free [class objectMeta 0x734c70] @ 0x734c70
free class @ 0x734c40
free [class class 0x734cc0] @ 0x734cc0
free object @ 0x734b00
free [class object 0x734ab0] @ 0x734ab0
free [module 0x734250] @ 0x734250
free [map 0x734110] @ 0x734110
[work@work b]$
```




仅供非商业用途或交流学习使用





自制编程语言

基于C语言

本书专家推荐：

现滴滴系统部技术高级总监 **于晓声**

阿里蚂蚁金服技术专家 **肖金亮**

百度资深运维工程师 **陈晓聪**

360 企业安全集团政企云事业部技术总监 **冯顾**

ACFUN 高级运维总监 **陆景玉**

Mobvista 运维总监 **黄梦溪**

异步社区 www.epubit.com

异步社区 www.epubit.com
新浪微博 @人邮异步社区
投稿/反馈邮箱 contact@epubit.com.cn

封面设计：董志桢

分类建议：计算机／操作系统开发
计算机／程序设计
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-48737-7

ISBN 978-7-115-48737-7
定价：89.00 元